Soft RISC-V Core for Education

By

Jack Parkinson (U1552044)

A dissertation presented to the School of Computing and Engineering

University of Huddersfield, UK.

In partial fulfillment of the requirements for the Final Year Project Module (NHP2400)

Academic Year: 2018-2019

By submitting this work I confirm that it is entirely my own and adheres to all the academic integrity guidelines set out by the University.

Abstract

RISC-V is a free and open source ISA that is still in the early stages of building its community. This project aims to accelerate that process by designing a RISC-V microprocessor core, which could be used as a tool to educate people on computer architecture and organisation. This dissertation will detail the design itself, a VHDL implementation of the design, and the simulation results of the VHDL implementation. Overall, the project was a success as the final design was found to serve as a useful education tool. It demonstrates many microarchitecture implementation techniques while still being relatively simple in its overall design. However, before it can be made open source additional optimisations need to be made, to make the VHDL implementation useable on FPGAs.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

Abbreviation	Description
RISC	Reduced Instruction Set Computer
CISC	Complex Instruction Set Computer
ISA	Instruction Set Architecture
FPGA	Field Programmable Gate Array
RTL	Register-Transfer Level
I/O	Input/Output
VLIW	Very Long Instruction Word
Mux	Multiplexer
Reg	Register
PC	Program Counter
ALU	Arithmetic & Logic Unit
CSR	Control & Status Register
BRAM	Block Random Access Memory
EDA	Electronic Design Automation
PC	Program Counter

1 Introduction

1.1 Background

Emerging technologies such as IoT, Machine Learning and Blockchain are driving new growth in the industry. While the functionality of these technologies continues to evolve, their practicality can't keep pace. This mainly comes down to the design of the physical devices used for these technologies. At the centre of nearly all these designs, there has been a general-purpose microprocessor. To meet the performance and efficiency needs of these emerging technologies, more application specific microprocessor designs are needed. For example, the microprocessors used in most IoT devices may only need to communicate with one I/O device, such as a wireless transmitter/receiver, and thus does not need a complex I/O module that would simply waste energy.

General-purpose microprocessors have dominated the market as economies of scale do not work in favour of these more application specific designs. Over the past decade, the software industry has managed to alleviate economies of scale with the advance of the Open Source model. RISC-V is a free and open ISA that aims to replicate the success of the open source seen in the software industry, at the hardware level.

Typically, what makes an open source project successful is the community it builds around itself. The larger and more active the community, the more contributions it will receive, leading to long term development and support. RISC-V is still in the early stages of building its community and seems to be primarily focused on industry. A tried and true method for building a community around a platform is to educate people on that platform. This was the fundamental concept upon which this project was based.

1.2 Aim

Design a microprocessor core that demonstrates common microarchitecture techniques, utilises the RISC-V ISA and is simple enough that it could be used as an educational tool in computer architecture and organisation.

1.3 Objectives

1. Design a soft processor core capable of executing Integer Computation, Load/Store and Control Transfer instructions
2. Implement the soft processor core using VHDL for use in FPGAs
3. Test the implementation in simulation
4. Test the implementation on a FPGA development board

1.4 Deliverables

1. RISC-V core design
2. Synthesisable VHDL code
3. Simulation test results

2 Research

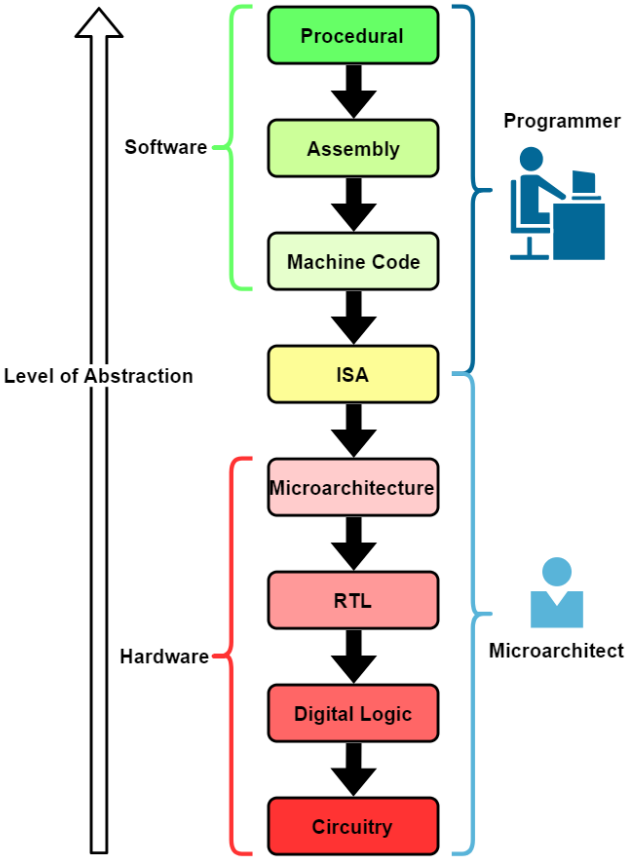
In this section, the theory required to understand the design choices made for this project will be explained and discussed where necessary. There are two main aspects to this theory, ISAs and Microarchitecture. As this project adheres to the RISC-V ISA, it will not discuss in detail the trade-offs made when designing an ISA. However, in section 2.1, the key components that make up an ISA will be explained. This is done as a firm understanding of ISAs is required to fully understand the design decisions made in section 3. In section 2.2, many of the common implementation techniques used in microarchitecture will be explained and trade-offs of each technique discussed.

2.1 Instruction Set Architectures

2.1.1 What is an ISA?

An ISA is a layer of abstraction that separates software from hardware in a machine. It details the binary instructions that a piece of software must compile down to, to run on a specific piece of hardware. This hardware that has been designed around an ISA is known as an implementation. Software that adheres to an ISA will typically run on any implementation of that ISA. If ISAs did not exist, then every piece of software would be specific to a single implementation and would have to be re-written to run on any other implementation.

Figure 2.1:1 - Computer Abstraction Model



The job of the ISA is to define everything that a machine level programmer would need to program an implementation. Anything that does not need to be exposed to the programmer is excluded from the ISA and left to the microarchitect (implementation designer). What exactly is defined varies between ISAs but generally, they will at least define the:

- Instructions
- Operands
- Registers
- Memory architecture

2.1.2 Instructions

The instructions are the set of binary strings that can be recognised and executed by a machine, the full instruction set could be thought of as the machine's language. These instructions can be encoded in numerous ways; however, they can usually be characterised by:

- **Operations**
  These are the types of instructions that can be performed by the machine e.g. integer arithmetic and logic, data transfer, control transfer, floating point arithmetic, etc. The operation of each instruction in an instruction set is identified by its Opcode.
- **Instruction length**
  This is how many bits make up a single word (instruction). The most common lengths are 8, 16, 32 & 64 bits but there are ISAs that use less than 8 bits. There are also some ISAs that use far more than 64 bits per word and are classified as VLIW ISAs. When the length of all the instructions in an instruction set is constant, it is known as fixed length encoded. A good example of a fixed length encoded instruction set is MIPS *[figure 2.1:2]*. When the length of the instructions in an instruction set varies, usually based on the opcode, it is known as variable length encoded. A good example of a variable length encoded instruction set is x86 *[figure 2.1:3]*.
- **Uniformity**
  Uniform decode is when the same bits in an instruction represent the same thing e.g. bits 31-26 in MIPS *[figure 2.1:2]* always represents the opcode. Non-uniform decode is the opposite e.g. bits 0-7 could be the opcode or instruction prefix in x86 *[figure 2.1:3]*.

Figure 2.1:2 - MIPS32 Instruction Formats [1, p.74]

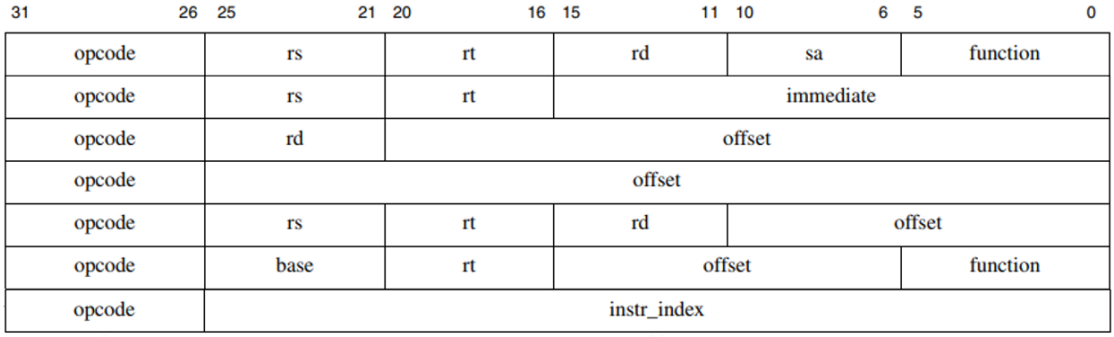
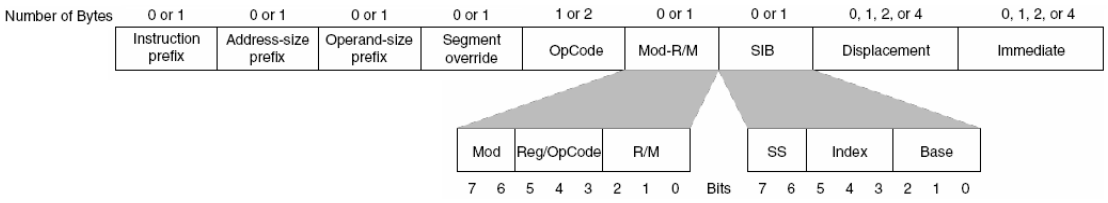


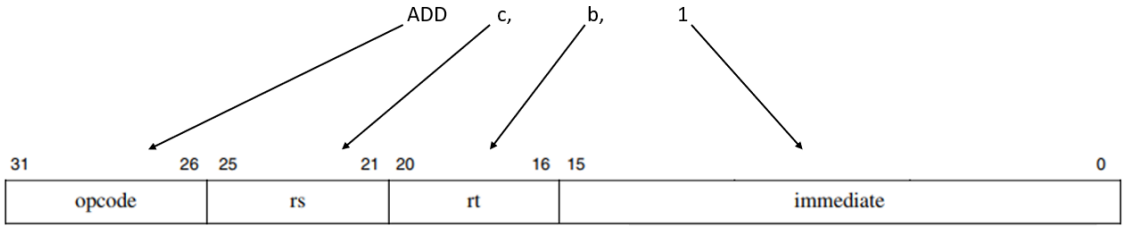
Figure 2.1:3 - x86 Instruction Format [2]



2.1.3 Operands

In an instruction, the operands are the data to be operated on. This comes from mathematics where, in the arithmetic expression '$1 + b = c$', '1' and 'b' would be the operands. In assembly, the same expression can be written as '$ADD\ c, b, 1$' where again '1' and 'b' would be the operands. Mapping this to a MIPS instruction *[figure 2.1:4]* would yield 'rs' and '$immediate$' as the operands. The term Immediate simply refers to an operand that is baked directly into an instruction. If the operand is not an immediate then it could be the contains of a register or the contains of a location in memory.

Figure 2.1:4 - Assembly to MIPS



Typically, the ISA will define the:

- **Types of operands**
  These are the data types that the hardware can compute, the most common types are:
  - Integer
  - Unsigned Integer
  - Single precision floating point
  - Double precision floating point

  An ISA can support any combination of data types but nearly all support Integer and Unsigned Integers. Data types can also be simulated at the software level.
- **Size of operands**
  This is how many bits make up the operand, the most common sizes are:
  - Byte (8 bits)
  - Half-word (16 bits)
  - Word (32 bits)
  - Double word (64 bits)

  It's worth noting that "word" can refer to 32 bits or the length of the instruction e.g. a word in a fixed length 64-bit Instruction set could mean 64 bits instead of 32 bits.
- **Number of operands**
  This is how many operands can be addressed in a single instruction and is also referred to as how many addresses the machine supports. In particular, it refers to the maximum number of operands the machine can address, as some instructions within the same ISA may pack more operands into them than others. The most common configurations are:
  - Zero-address machines (0 operands per instruction)
  - One-address machines (up to 1 operand per instruction)
  - Two-address machines (up to 2 operands per instruction)
  - Three-address machines (up to 3 operands per instruction)

2.1.4 Registers

Most ISAs will detail two types of data storage, registers, and memory. Registers are the fastest and often smallest form of data storage. Anything stored in a register will normally be processed as quickly as possible, and then sent back to memory. Usually, an ISA will only expose registers that store operands to the programmer. These can either be explicit (directly referenced in the instruction) or implicit (directly referenced in the instruction). How the programmer should handle these registers depends heavily on the number of operands. The most common configurations are:

- **Stack Machine**
  These are zero-address machines where the registers are configured as a stack. The operands are implicitly located on either the top or the bottom of the stack.
- **Accumulator Machine**
  These are one-address machines where intermediate operands are stored in a register (the accumulator). One operand is explicitly referenced in the instruction and the other is implicitly located in the accumulator.
- **General-Purpose Register Machine**
  These are two or more address machines where all operands are explicitly referenced. Nearly all modern processors fall into this category as it allows the programmer the most flexibility. Most of these machines fall into one of two categories:

Table 2.1:1 - Register Machines [3, p.8]

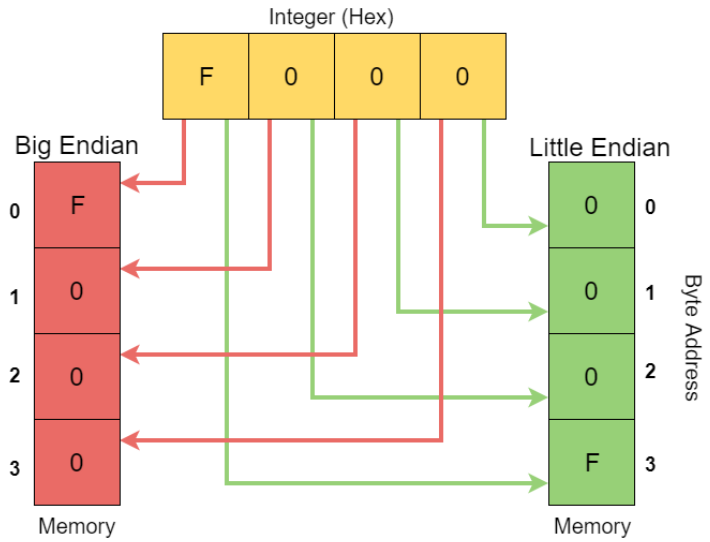
Machine type	Description	Number of memory addresses	Number of operands
Register-Register (Load/Store)	Operands can only be operated on from registers	0	2-3
Register-Memory	Operands can be operated on from memory or registers	1-2	1-2

2.1.5 Memory Architecture

There are many aspects to the memory architecture of a machine that an ISA may define. Nearly all ISAs will define:

- **Byte Order**
  This is the order of data stored in memory which can either be Little Endian or Big Endian. Little Endian is where the least significant bits are stored first. Big Endian, on the other hand stores the most significant bits first.

Figure 2.1:5 - Endianness



- **Byte Alignment**
  An ISA may or may not allow misaligned access. Misaligned access allows the programmer to address any byte location in memory. Aligned access only allows the programmer to address bytes locations that are a multiple of the data type's length. For example, storing a 32-bit (4-byte) word in memory address 4 would be aligned *[table 2.1:2]*.

Table 2.1:2 - Word Alignment [3, p.16]

Mem Addr	0	1	2	3	4	5	6	7
	Aligned				Aligned			
		Misaligned				Misaligned		
			Misaligned				Misaligned	
				Misaligned				Misaligned

- **Addressing Modes**
  When an operand is not an immediate, the addressing mode determines how the effective address is computed, whether it be in memory or registers. The most common addressing modes in general-purpose register machines are:

Table 2.1:3 - Common Addressing Modes

Addressing Mode	Example Assembly	RTL
Immediate	$ADD\ R1, 4$	$Regs[R1] = Regs[R1] + 3$
Absolute	$ADD\ R1, (4)$	$Regs[R1] = Regs[R1] + Mem[4]$
Register	$ADD\ R1, R2$	$Regs[R1] = Regs[R1] + Regs[R2]$
Register Indirect	$ADD\ R1, (R2)$	$Regs[R1] = Regs[R1] + Mem[Regs[R2]]$
Displacement	$ADD\ R1, 4(R2)$	$Regs[R1] = Regs[R1] + Mem[4 + Regs[R2]]$
Indexed	$ADD\ R1, (R2 + R3)$	$Regs[R1] = Regs[R1] + Mem[Regs[R2] + Regs[R3]]$
Memory Indirect	$ADD\ R1, @(R2)$	$Regs[R1] = Regs[R1] + Mem[Mem[Regs[R2]]]$

2.1.6 RISC vs CISC

There are many ways that an ISA may be categorised as seen in sections 2.1.2-2.1.5, however, most ISAs traditionally fell into one of two categories. These categories are CISC and RISC. To understand the fundamental differences between these two categories, the concept of the semantic gap must be understood. The semantic gap is how closely an ISA parodies programming languages. The more closely instructions in an ISA resemble high-level code, the smaller the semantic gap. CISC ISAs aimed to reduce the semantic gap whereas RISC ISAs increased the semantic gap, to reduce hardware complexity. There are common traits of a RISC vs CISC ISA *[table 2.1:4]*. Nearly all ISAs will have a combination of RISC and CISC traits, but will have more of one than the other. For example, an ISA may have all the traits of a CISC ISA but have a load-store architecture.

Table 2.1:4 - RISC vs CISC

Traits	RISC	CISC
Instructions	Few operations	Many operations
	Fixed length	Variable length
	Uniform decode	Non-uniform decode
Operands	Few data types	Many data types
	2-3 operands per instruction	3 or more operands per instruction
Registers	Load-Store	Register-Memory
	Many registers	Few registers
Memory Architecture	Aligned access	Misaligned access
	Few addressing modes	Many addressing modes

Throughout the 1970s, 1980s and 1990s CISC implementations dominated the market thanks to their more efficient use of memory (variable length instructions) and silicon area (fewer registers). RISC implementations gained populate in the 2000s thanks to ever-shrinking manufacturing processes. These smaller manufacturing processes allowed for higher transistor counts and memory capacity. This negated many of the advantages CISC had over RISC. Nowadays most CISC implementations utilise microcode meaning they are essentially RISC designs at their lowest level. Microcode is discussed in more detail in section 2.2.2.

2.1.7 RISC-V

Having covered the relevant theory regarding ISA design, let's examine RISC-V and discuss why it was chosen for this project. While RISC-V is not the first open-source ISA, it is the first designed to be user extensible without breaking existing extensions or incurring software fragmentation. It is comprised of base integer instruction sets and optional extensions. There are multiple base integer instruction sets to allow for varying length instructions (32, 64 and 128-bit). A hardware implementation may only implement one of these, in which case all instructions would be of a fixed length. The optional extensions allow the functionality of the base integer instruction set to be expanded upon. This helps to keep the ISA itself relevant by allowing future technologies to be added onto the base integer ISA. Existing standard extensions include:

- `M` – Integer multiplication and division
- `A` – Atomic Instructions
- `F` – Single-precision floating-point
- `D` – Double-precision floating-point
- `Q` – Quad-precision floating-point
- `L` – Decimal floating-point
- `C` – Compressed Instruction
- `B` – Bit manipulation
- `J` – Dynamically translated languages
- `T` – Transactional memory
- `P` – Packed-SIMD instructions
- `V` – Vector operations
- `N` – User-level interrupts

The base integer instruction sets along with some of the existing standard extensions have been frozen. Frozen simply means that the specification for the ISA will not change from a certain point in time. Being open-source, this is very important as it prevents software fragmentation. Otherwise, software developers would have no reassurance that what they've developed would work on future hardware.

The primary reason RISC-V was chosen for this project is the free and open-source model RISC-V adopts. This will allow students to use a variety of tools developed by the open-source community e.g. compilers, simulators, etc. Also, it will provide students with a wide array of supporting material outside of the scope of this project, should they wish to progress further. The ISA being a frozen is important for an educational tool as it ensures that it will be compatible with future software and will, therefore, remain relevant. This should give students more confidence in what they're learning as it will be applicable in the industry for years to come.

2.2 Microarchitecture

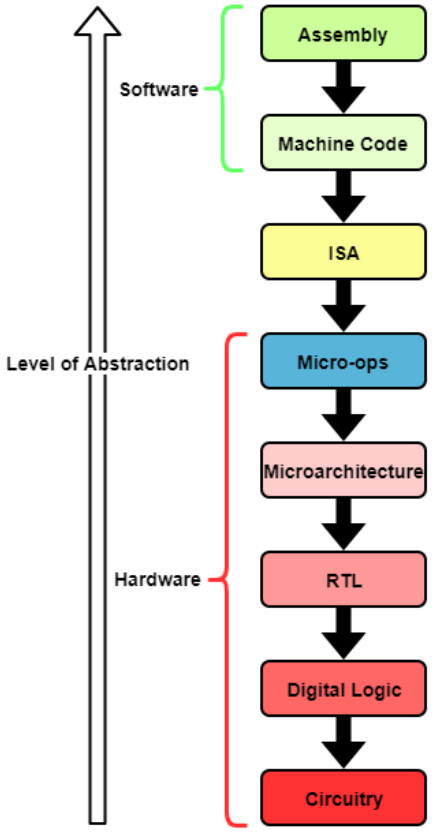
2.2.1 What is Microarchitecture?

Microarchitecture is the implementation of an ISA in hardware under specific design constraints and gaols *[4, p.8]*. It is considered the highest level of computer hardware abstraction *[figure 2.1:1]* and mainly deals with how data flow and/or control flow is architected. Instead of focusing on improving the performance of individual execution blocks, microarchitecture instead deals with optimising the flow of data throughout the entire implementation. This can be optimising for throughput, latency, power consumption or some combination of the aforementioned goals. Most details of the microarchitecture are typically made invisible to the programmer.

2.2.2 Microcode

Microcode is an implementation technique used mainly in CISC designs, to break up complex instructions into multiple RISC style micro-operations. This introduces another translation layer which separates the assembly instructions visible to programmers from the machine level instructions executed by hardware. This translation layer is typically implemented at the hardware level. This drastically simplifies the hardware implementation, which in turn makes it easier for the microarchitect to better optimise the design to meet the desired design constraints and goals. It also allows the ISA to be expanded upon and become more complex, without having to worry about how it will directly affect hardware implementations. The main downside is that additional silicon area is required to implement this translation layer. A microcoded CISC implementation therefore consumes more power than a RISC implementation.

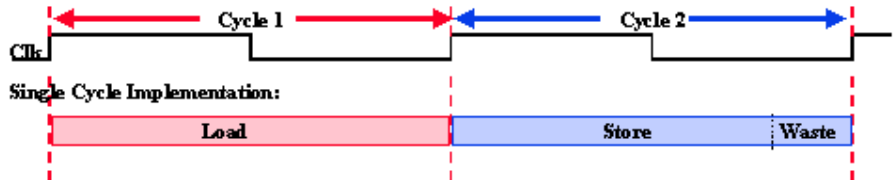
Figure 2.2:1 – Microcode abstraction model



2.2.3 Single-Cycle vs Multi-Cycle vs Pipelined

There are three main approaches to processing an instruction with respect to the clock. The first is to process each instruction in a single clock cycle, this is known as a single-cycle machine *[figure 2.2:2]*. These are the simplest machines to implement as they consist of just two states and pure combinational logic to process the instruction. However, they have the slowest clock speed as it is determined by the instruction which takes the longest time to execute (critical path). Single-cycle machines are useful for applications where reduced silicon area and power consumption are more important than performance.

Figure 2.2:2 - Single Cycle Machine [5]

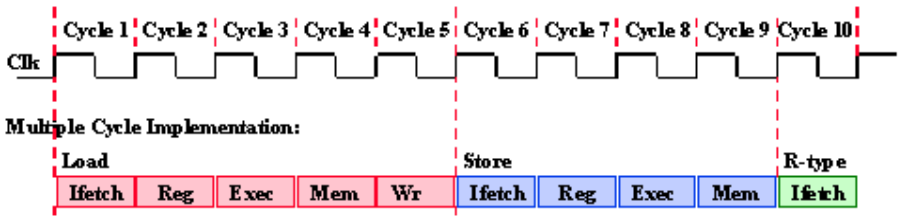


The second approach is to process each instruction in stages, this is known as a multi-cycle machine. In the example below *[figure 2.2:3]*, the instruction is broken up into the following stages:

1. **Ifetch** – Instruction fetch
2. **Reg** – Registers read
3. **Execute** – Execute
4. **Memory** – Memory access
5. **Write Back** – Register write back

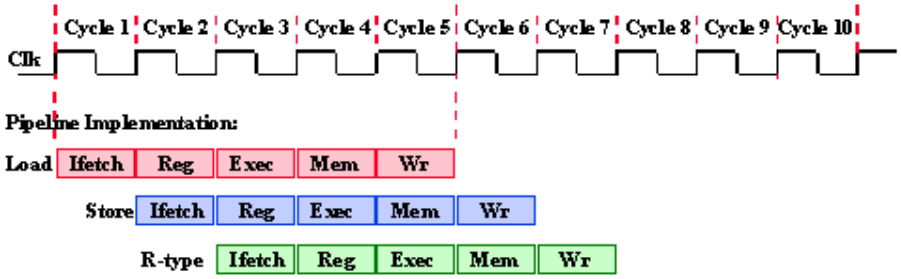
Multi-cycle machines are more complex machines to implement as they consist of multiple states and require sequential logic to process the instruction. The clock speed is much faster than a single-cycle machine as it is determined by the time taken to complete a single stage rather than processing the entire instruction. This does not reduce the critical path; in fact, it often increases it. However, it allows instructions that do not use certain stages to skip them entirely. This means that the time taken to process the most complex instructions may increase in a multi-cycle machine vs a single-cycle machine however, the average time to process an instruction will be reduced.

Figure 2.2:3 - Single Cycle Machine [5]



The third approach is identical to a specific form of multi-cycle machine, were instructions are overlapped. This means each stage processes a different instruction in parallel. This is known as a pipelined machine *[figure 2.2:4]*. This does not reduce the average time taken to process an instruction over a standard multi-cycle machine, but it increases throughput. Due to this increase in throughput, an entire program would execute much faster on a pipelined machine vs a multi-cycle machine.

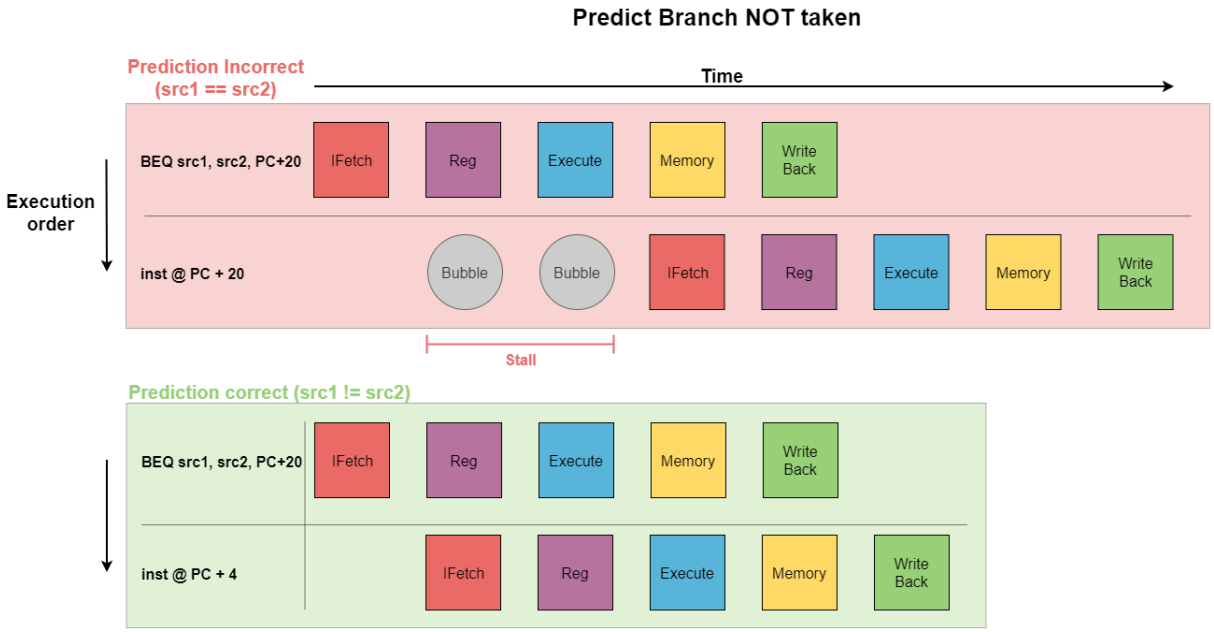
Figure 2.2:4 – Pipelined Machine [5]



2.2.4 Branch Prediction

Nearly all control flow programs do not execute instructions in a purely sequential manner, they will jump ahead/behind, or branch ahead/behind based on a condition. In a single-cycle or standard multi-cycle machine, a jump or branch does not affect performance. In a pipelined machine, a jump or branch causes instructions already in the pipeline to become meaningless to the proper execution of the program. This is known as a control hazard and causes a pipeline stall which negatively affects performance. The most common way of preventing pipeline stalls is with an implementation technique called branch prediction. The hardware predicts if a branch is taken or not, and then speculatively executes an instruction. If the prediction was accurate then there is little to no delay in the pipeline. However, if the prediction is inaccurate the pipeline will need to be stalled. Therefore, the effectiveness of this technique comes down to how accurate the microarchitect can make the predictions.

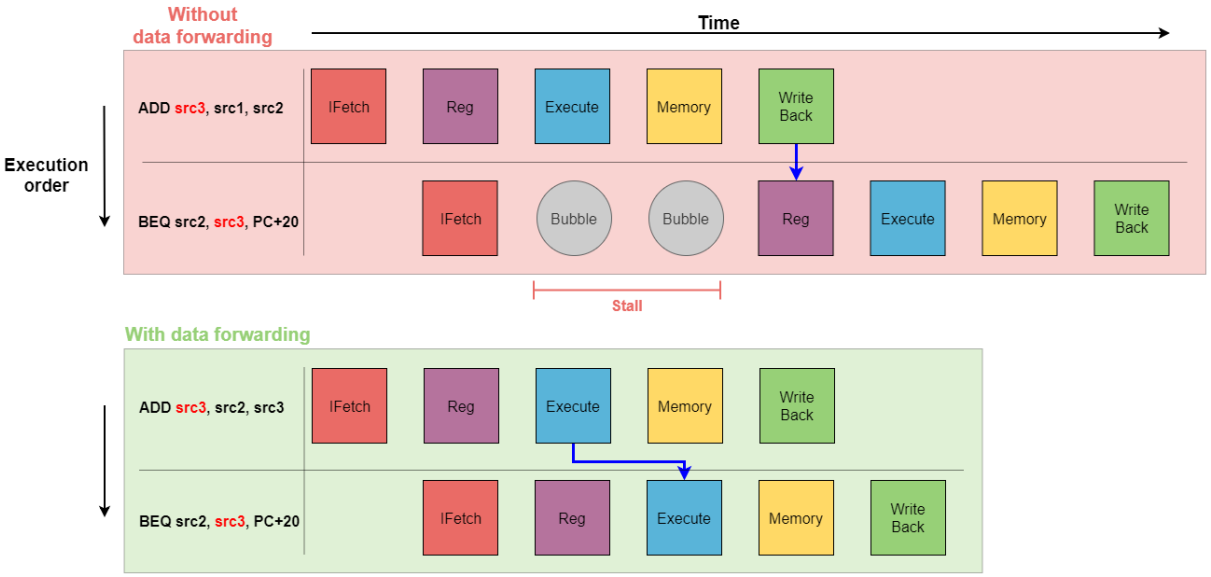
Figure 2.2:5 – Example Branch Prediction



2.2.5 Data Forwarding

Due to instructions being overlapped in execution in a pipelined machine, the order of read/write access to operands can be altered [6]. This is known as a data hazard, and again, it causes a pipeline stall negatively affecting performance. In this case, the most common way of preventing pipeline stalls is to forward the results from one pipeline stage to another.

Figure 2.2:6 - Example Data Forwarding



3 Design

The focus of this section will be to detail and discuss any microarchitecture design choices made. There will also be an overview of the ISA, but there will be no discussion on why certain design choices were made in the ISA itself. These are already well documented in the RISC-V user level specification [7].

3.1 Instruction Set Architecture

3.1.1 Base Integer Instruction Set

As discussed in section 2.1.7, the RISC-V ISA has multiple base integer instruction sets. Those are RV32I, RV32E, RV64I & RV128I, with the main difference between them being the instruction length. The aim of this project is to design an educational tool for people new to computer architecture and organization. RV32E would make for the simplest and most compact design and would, therefore, seem like the obvious choice. However, as of the time of writing this, RV32E and RV128I have not yet been frozen. Due to this RV32I was selected to ensure compatibility with future software, while still being simpler and more compact than an RV64I implementation.

Notable characteristics of RV32I include:

- 31 general-purpose registers, 1 constant zero register, and 1 program counter register. All of which are 32-bits wide and are visible to the programmer

Table 3.1:1 - Registers [7]

31	0
x0 / zero	
x1	
x2	
x3	
x4	
x5	
x6	
x7	
x8	
x9	
x10	
x11	
x12	
x13	
x14	
x15	
x16	
x17	
x18	
x19	
x20	
x21	
x22	
x23	
x24	
x25	
x26	
x27	
x28	
x30	
x31	
pc	

- Load/Store architecture
- Little Endian
- Uniform decode
- 32-bit integer (Two's compliant) and unsigned Integer operands
- Up to three operands per instruction
- Support for misaligned memory access
- Uses the following addressing modes:
    - Absolute
    - Register-offset
    - PC-relative
- Includes the following operations:
    - Integer Computation
    - Control Transfer
    - Load/Store
    - Hardware Thread Management (hart)
    - CSR
    - Environment Call and Breakpoints (Debugging)

Due to time constraints and the fact that they are not common to all ISAs, the hart, CSR and debugging instructions were excluded from this design of the implementation. The implications of this will be further discussed in section 7.2.

3.1.2 Instruction Format

RV32I has four core instruction formats (R/I/S/U) and two immediate encoding variants of the S/U formats which are B/J respectively. As mentioned in the previous section, RV32I uses uniform encoding and all instruction must be aligned on a 4-byte boundary in memory.

Table 3.1:2 - Instruction & Immediate Formats [7]

Key	
opcode	operation identifier
rd	address (x0-31) of register to write results back to (destination register)
rs1/2	address (x0-31) of register to read operands from (source register)
funct3/7	sub operation identifier (function)
imm	constant operand value (immediate)

31	30	25	24	21	20	19	15	14	12	11	8	7	6	0	
funct7				rs2		rs1		funct3		rd			opcode		R-type
imm[11:0]						rs1		funct3		rd			opcode		I-type
imm[11:5]				rs2		rs1		funct3		imm[4:0]			opcode		S-type
imm[12]	imm[10:5]			rs2		rs1		funct3		imm[4:1]		imm[11]	opcode		B-type
imm[31:12]										rd			opcode		U-type
imm[20]	imm[10:1]				imm[11]	imm[19:12]				rd			opcode		J-type

31	30	20	19	12	11	10	5	4	1	0	
- inst[31]-						inst[30:25]		inst[24:21]		inst[20]	I-imm
- inst[31]-						inst[30:25]		inst[11:8]		inst[7]	S-imm
- inst[31]-					inst[7]	inst[30:25]		inst[11:8]		0	B-imm
inst[31]	inst[30:20]		inst[19:12]		-0-						U-imm
-inst[31]-			inst[19:12]		inst[20]	inst[30:25]		inst[24:21]		0	J-imm

3.1.3 Opcodes

In the RV32I instruction set, bits 6-0 represent the opcode, but the lowest two bits are always set to 11. This allows for the optional instruction compression with the “C” extension.

Table 3.1:3 - Opcodes

Key
Integer Computation
Control Transfer
Load/Store

Opcode (inst [6-0])	Name	Type
0110111	LUI	U
0010111	AUIPC	U
0010011	OP-IMM	I
0110011	OP	R
1101111	JAL	J
1100111	JALR	I
1100011	BRANCH	B
0000011	LOAD	I
0100011	STORE	S

3.1.4 Implementation Instructions Set

For full details on each instructions this design will be able to execute *[table 3.1:4],* see the RISC-V user-level specification [7].

Table 3.1:4 - Instruction Set

imm[31:12]					rd	0110111	LUI
imm[31:12]					rd	0010111	AUIPC
imm[20]	imm[10:1]	imm[11]	imm[19:12]		rd	1101111	JAL
imm[11:0]			rs1	000	rd	1100111	JALR
imm[12]	imm[10:5]	rs2	rs1	000	imm[4:1] imm[11]	1100011	BEQ
imm[12]	imm[10:5]	rs2	rs1	001	imm[4:1] imm[11]	1100011	BNE
imm[12]	imm[10:5]	rs2	rs1	100	imm[4:1] imm[11]	1100011	BLT
imm[12]	imm[10:5]	rs2	rs1	101	imm[4:1] imm[11]	1100011	BGE
imm[12]	imm[10:5]	rs2	rs1	110	imm[4:1] imm[11]	1100011	BLTU
imm[12]	imm[10:5]	rs2	rs1	111	imm[4:1] imm[11]	1100011	BGEU
imm[11:0]			rs1	000	rd	0000011	LB
imm[11:0]			rs1	001	rd	0000011	LH
imm[11:0]			rs1	010	rd	0000011	LW
imm[11:0]			rs1	100	rd	0000011	LBU
imm[11:0]			rs1	101	rd	0000011	LHU
imm[11:5]	rs2		rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2		rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2		rs1	010	imm[4:0]	0100011	SW
imm[11:0]			rs1	000	rd	0010011	ADDI
imm[11:0]			rs1	010	rd	0010011	SLTI
imm[11:0]			rs1	011	rd	0010011	SLTIU
imm[11:0]			rs1	100	rd	0010011	XORI
imm[11:0]			rs1	110	rd	0010011	ORI
imm[11:0]			rs1	111	rd	0010011	ANDI
0000000	shamt		rs1	001	rd	0010011	SLLI
0000000	shamt		rs1	101	rd	0010011	SRLI
0100000	shamt		rs1	101	rd	0010011	SRAI
0000000	rs2		rs1	000	rd	0110011	ADD
0100000	rs2		rs1	000	rd	0110011	SUB
0000000	rs2		rs1	001	rd	0110011	SLL
0000000	rs2		rs1	010	rd	0110011	SLT
0000000	rs2		rs1	011	rd	0110011	SLTU
0000000	rs2		rs1	100	rd	0110011	XOR
0000000	rs2		rs1	101	rd	0110011	SRL
0100000	rs2		rs1	101	rd	0110011	SRA
0000000	rs2		rs1	110	rd	0110011	OR
0000000	rs2		rs1	111	rd	0110011	AND

3.2 Data Flow

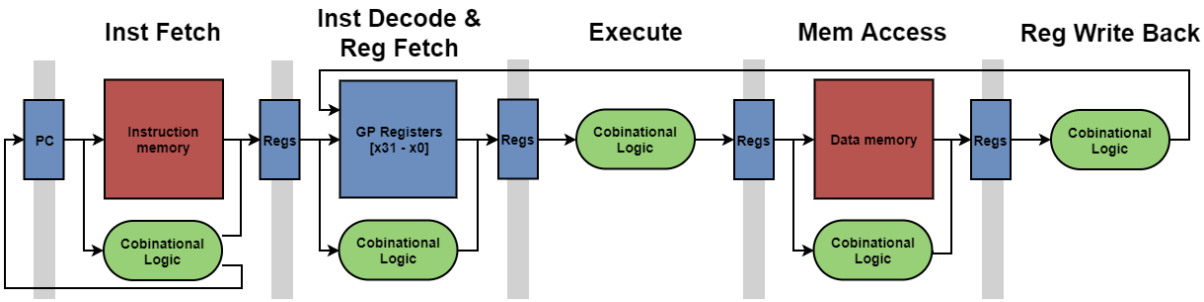
3.2.1 Pipeline Stages

Nearly all modern implementations, apart from the most compact cores, use a pipeline. For this reason, most computer architecture & organisation courses will cover pipelining in great detail. Therefore, to be a useful education tool this design will use a pipeline. This has the added benefit of making the core design compatible with more advanced implementation techniques such as out-of-order execution, superscalar, multithreading, etc. This allows students wishing to progress past the scope of this project, to expand upon this design.

In keeping with this mentality, this design will use a classic five-stage RISC pipeline as nearly all educational material on pipelining will cover it. The purpose of each stage is as follows:

1. **Inst Fetch** - fetch the next instruction from memory and increment the program counter
2. **Inst Decode & Reg Fetch** – decode the instruction and fetch operands from general-purpose registers
3. **Execute** – execute the instruction
4. **Mem Access** – load/store data into/out of memory
5. **Reg Write Back** – write any results back into the general-purpose registers

Figure 3.2:1 - Pipeline Overview

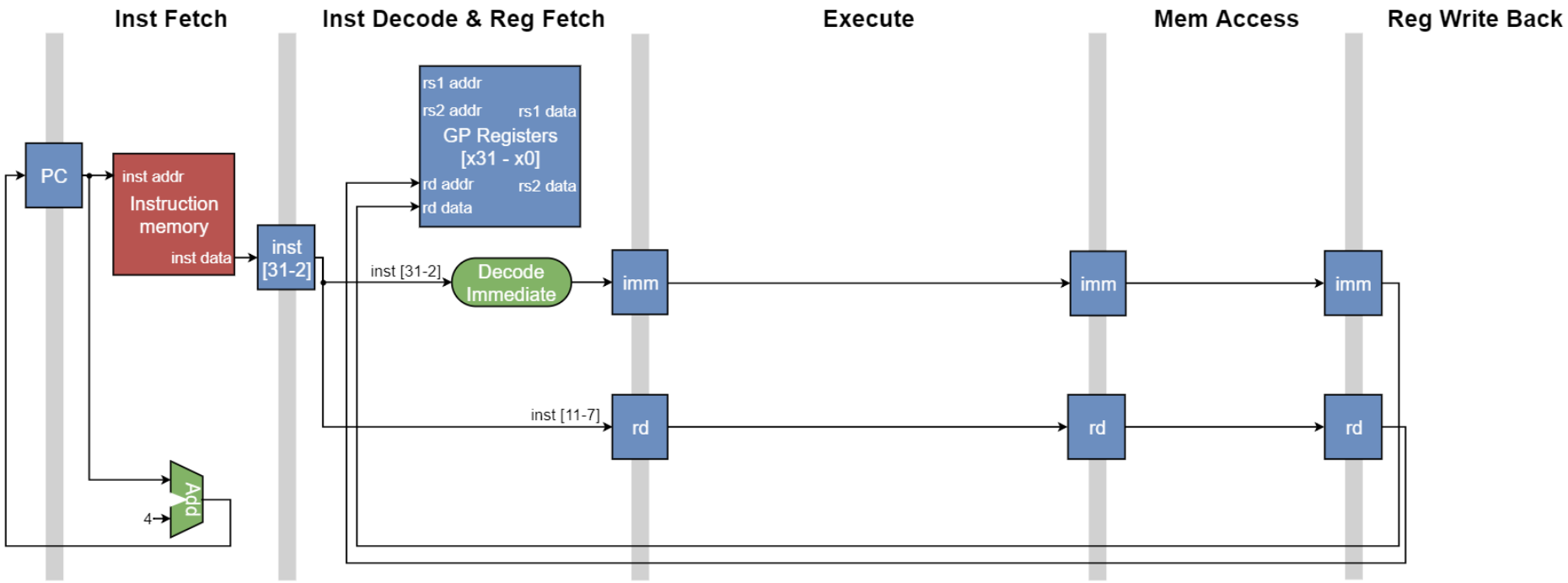


3.2.2 Opcode Specific Data Flows

The flow of data through the system will be determined by the opcode. To begin the design process, the data flow for each individual opcode was designed. These individual data flows will then be combined to form the full data flow in section 3.2.3. This was done to simplify the design process and minimise the chances of making mistakes in the full data flow for the core.

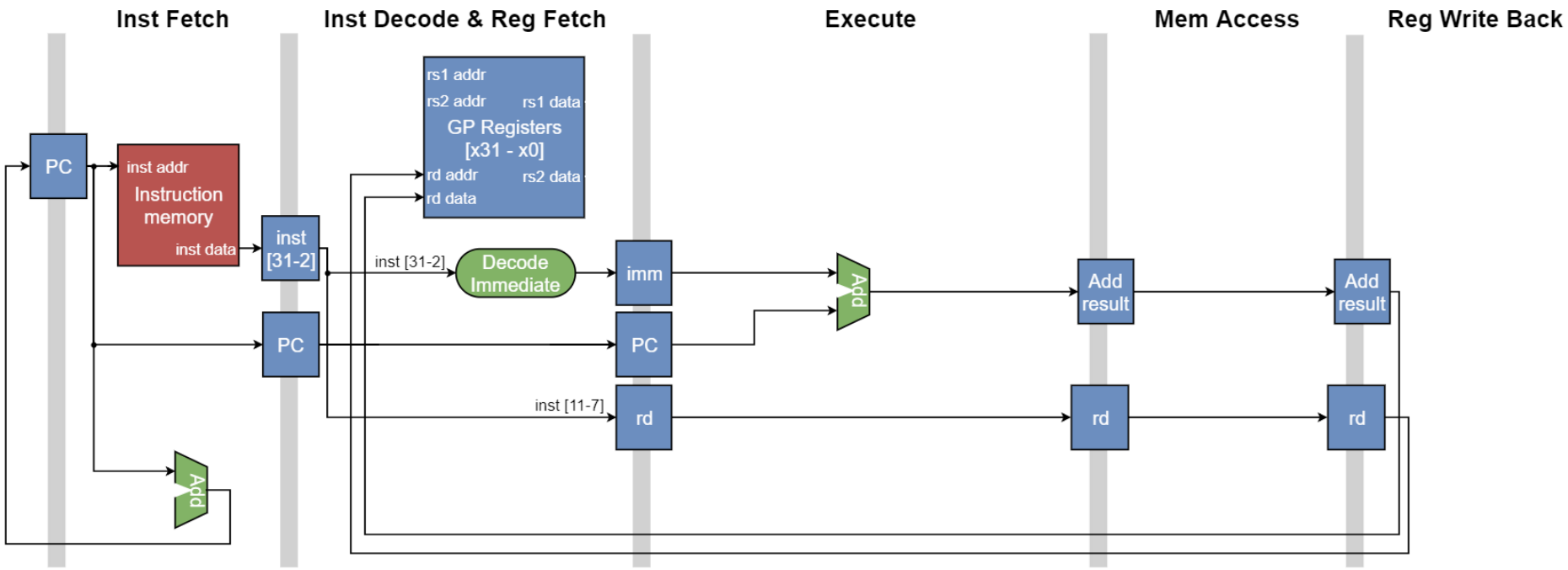
The LUI instruction is used to build constants and uses the absolute addressing mode. The immediate is simply decoded and then written into the destination register. It does not use any logic in the execute, mem access or reg writeback stages. However, the instruction must propagate through the entire pipeline to maintain the proper execution order.

Figure 3.2:2 - LUI Data Flow



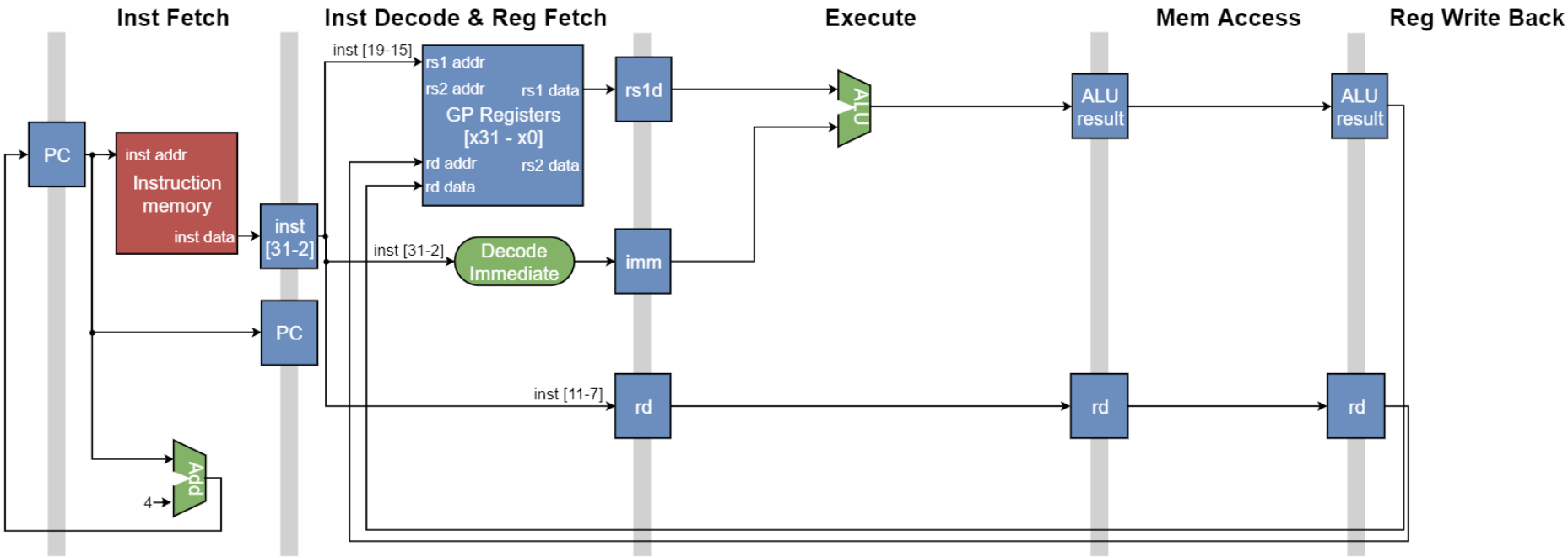
The AUIPC instruction is used to build constants and uses the PC relative addressing mode. The immediate is decoded, added to the current program counter and then written into the destination register. Again, it must propagate through the entire pipeline even though it does not use any logic in the mem access or reg writeback stages.

Figure 3.2:3 - AUIPC Data Flow



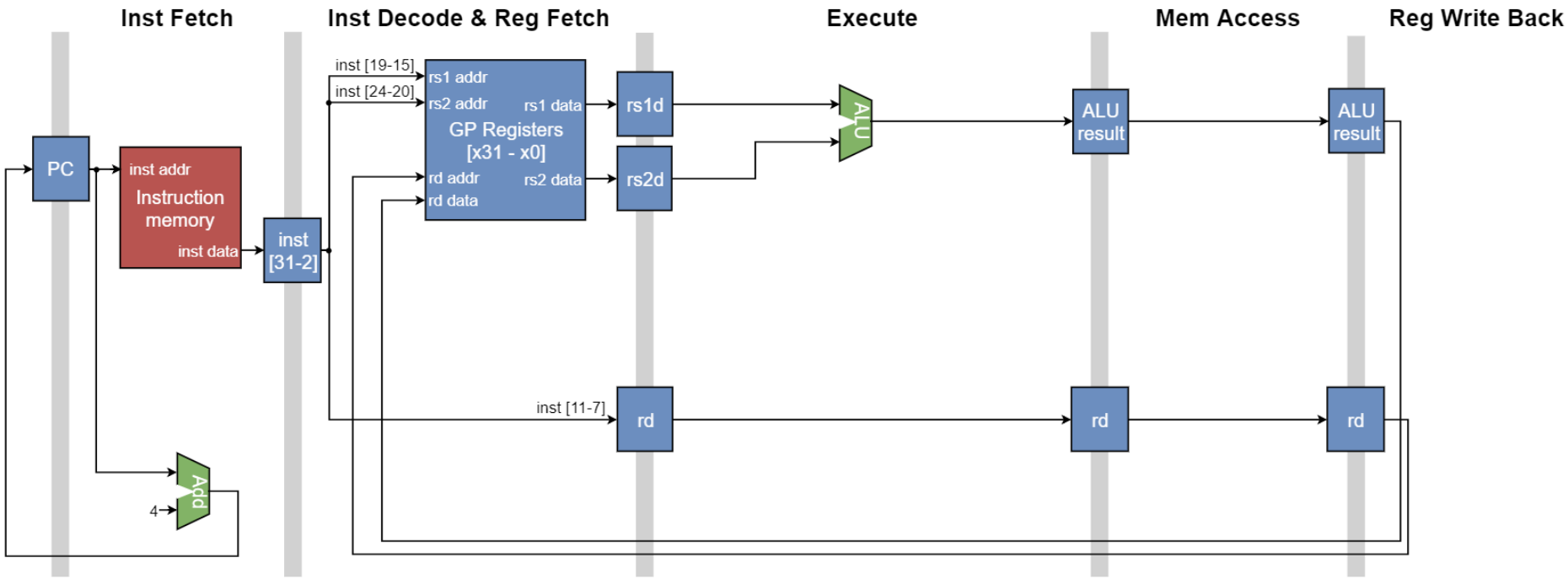
Instructions that use the OP-IMM opcode perform integer arithmetic on two operands, one immediate, and one register. The immediate is decoded while the value in the source register is fetched. The two operands are then fed into the ALU before being written into the destination register.

Figure 3.2:4 - OP-IMM Data Flow



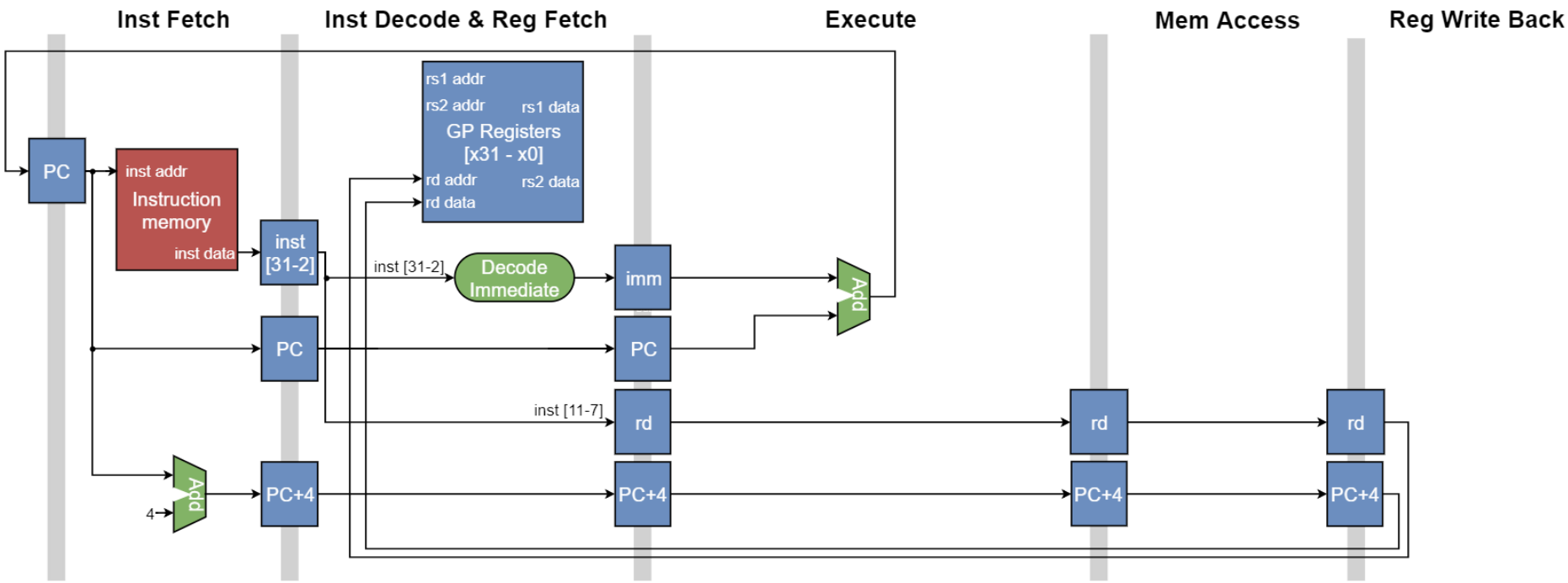
Instructions that use the OP opcode perform integer arithmetic on two register operands. The values in the two source registers are fetched, fed into the ALU and written into the destination register.

Figure 3.2:5 - OP Data Flow



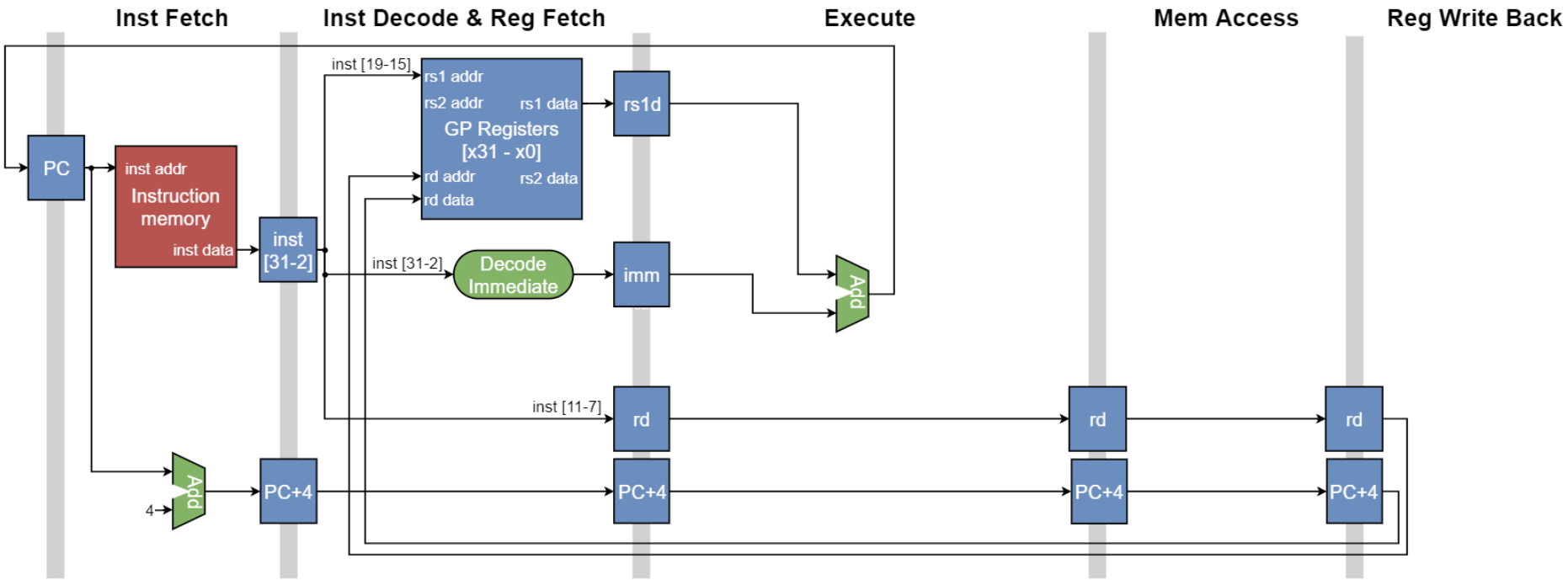
The JAL instruction is used to alter the program counter and uses the pc relative addressing mode. The immediate is decoded, added to the program counter, written into the program counter register and the address of the instruction following the jump (PC+4) is written into the destination register.

Figure 3.2:6 - JAL Data Flow



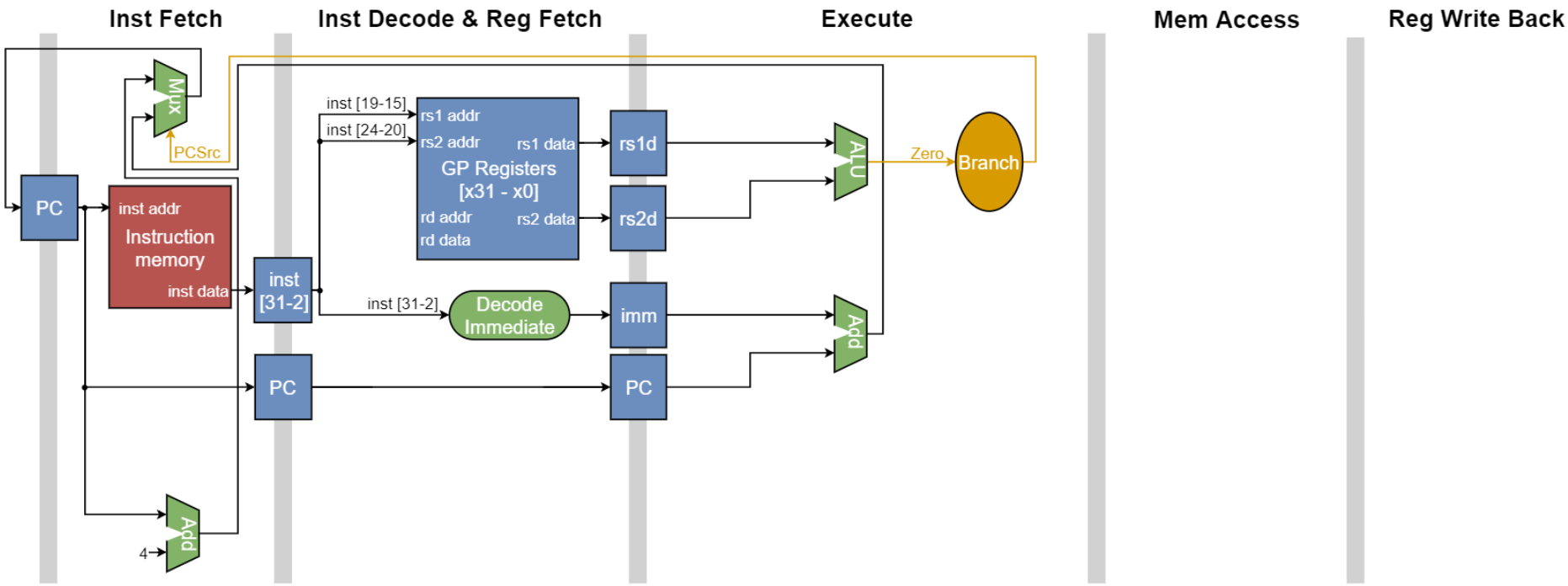
The JALR is used to alter the program counter and uses the register-offset addressing mode. The immediate is decoded while the value in the source register is fetched. These values are added together, written into the program counter register and the address of the instruction following the jump (PC+4) is written into the destination register.

Figure 3.2:7 - JALR Data Flow



Instructions that use the BRANCH opcode alter the program counter if a certain condition is met. The immediate is decoded while the values in the two source registers are fetched. The values fetched from the two source registers are fed into the ALU while the immediate is added to the program counter. If the result from the ALU is zero, then the result from the addition is written into the program counter.

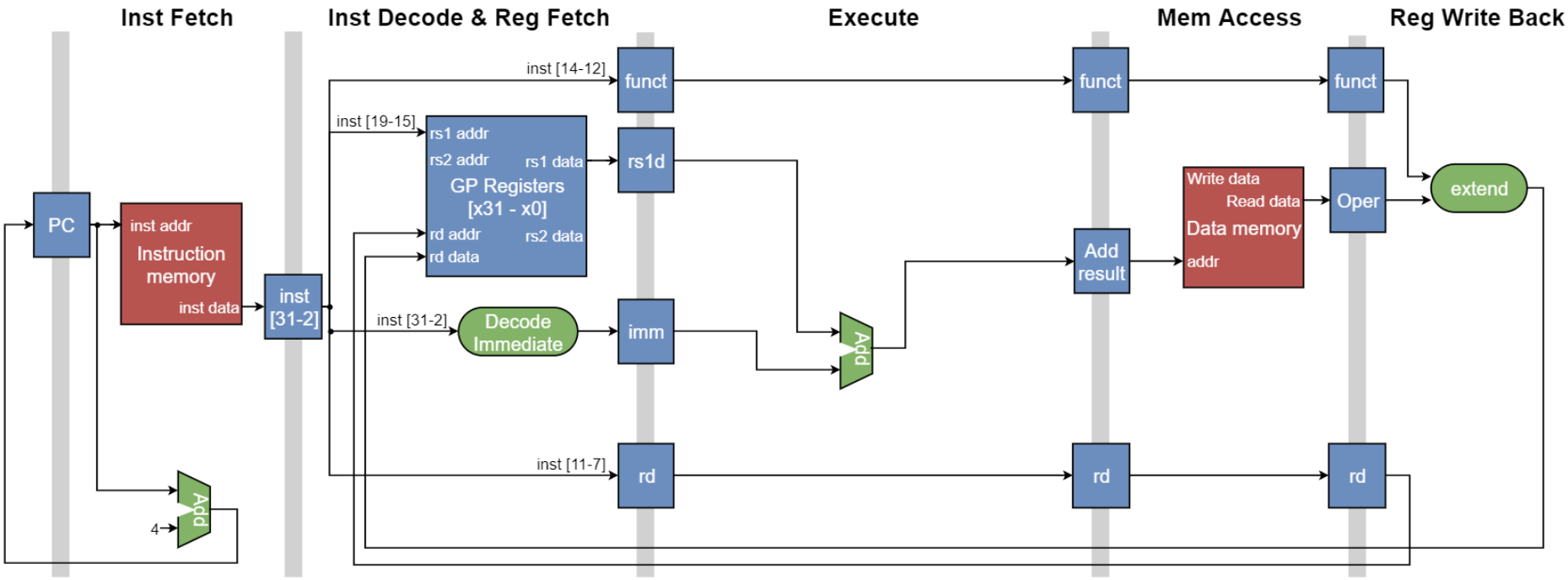
Figure 3.2:8 - BRANCH Data Flow



Here, the branch logic could have been placed in the mem access stage to reduce the critical path through the execution stage. However, it was placed in the execution stage to reduce the delay incurred when the branch is taken and the pipeline needs to be flushed. Also, the mem access stage is still very likely to be the slowest stage to execute, so the critical path through the execution stage won't determine the clock speed.

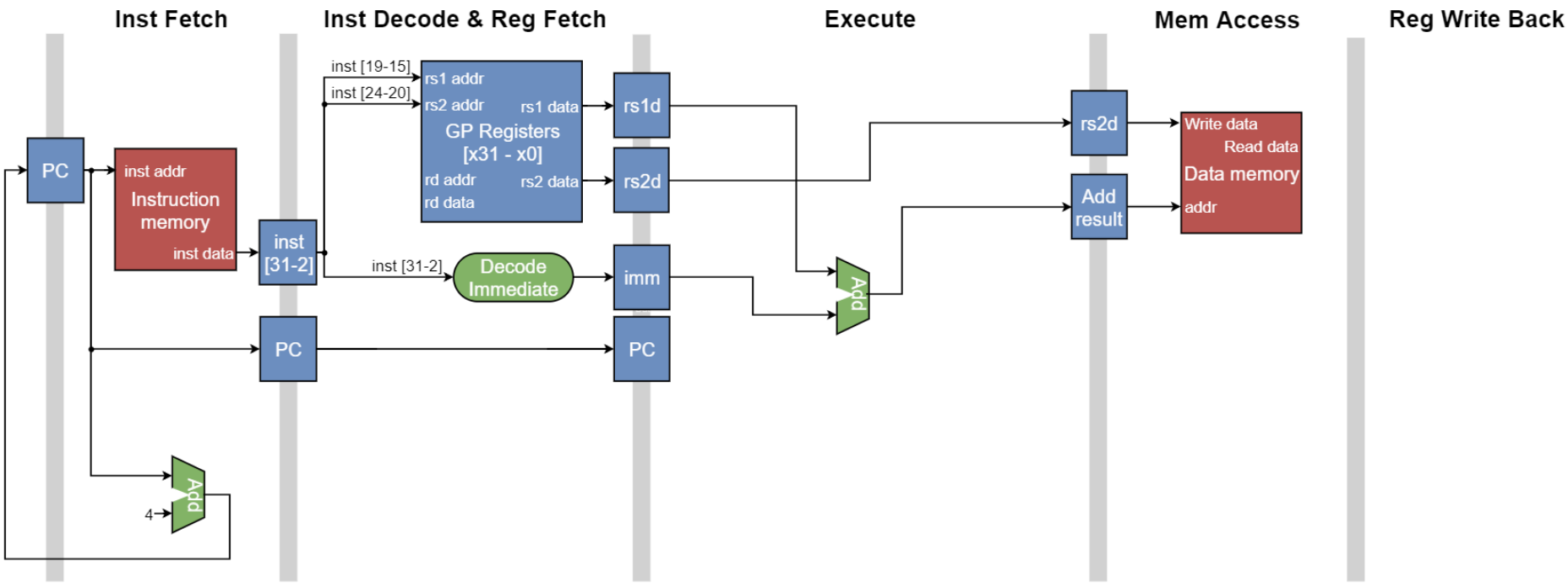
Instructions that use the LOAD opcode load data from memory into a register, using the register-offset addressing mode. The immediate is decoded while the value in the source register is fetched and added together, forming the memory address. The data in that address is loaded from memory, extended to 32-bits and written into the destination register.

Figure 3.2:9 - LOAD Data Flow



Instructions that use the STORE opcode store the value from a register into memory, using the register-offset addressing mode. The immediate is decoded while the value in the source registers is fetched and added together to form the memory address. That address is then used to store the data from the other source register into memory.

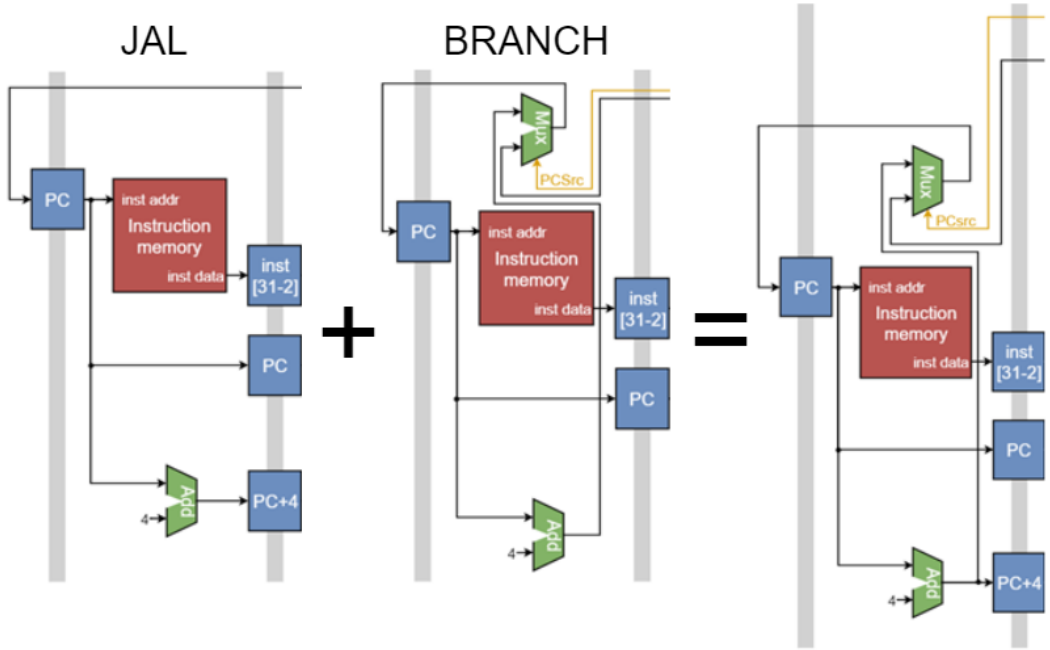
Figure 3.2:10 - STORE Data Flow



3.2.3 Full Data Flow

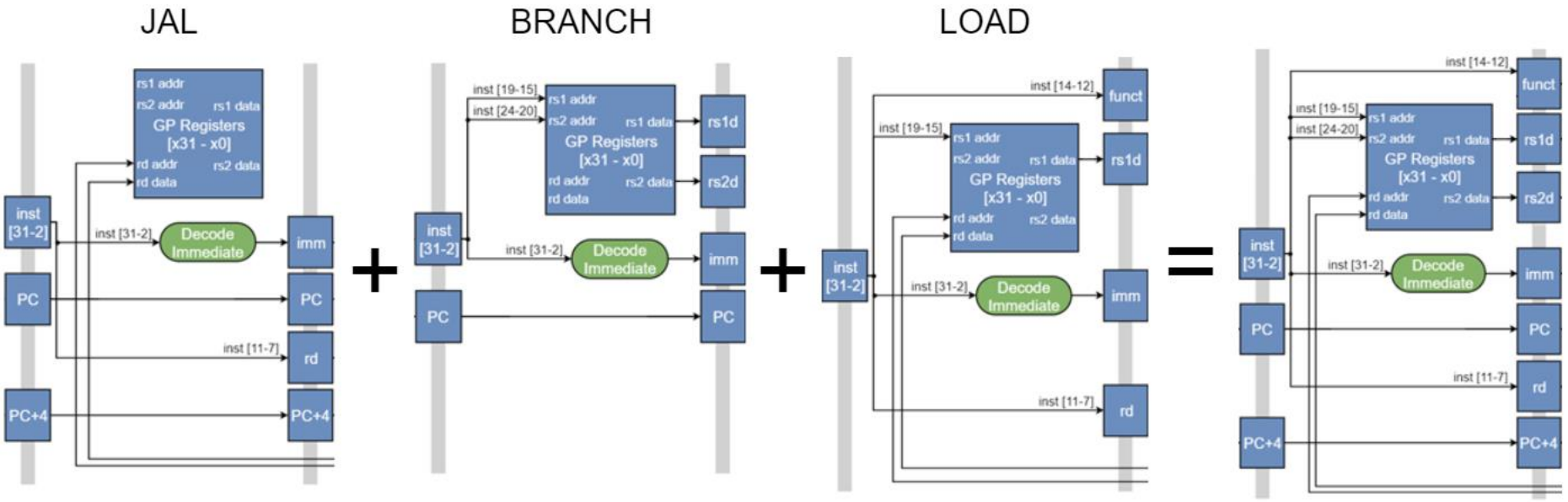
To combine all the data flows for each opcode into one data flow for the entire core, the inputs, and outputs from each stage were observed. For the Inst Fetch stage, the JAL data flow used the largest number of pipeline registers, so it was used as the base for the core data flow. The BRANCH data flow was the only one to use logic blocks not present in the JAL data flow, so it was superimposed on top of the JAL data flow to create the core data flow.

Figure 3.2:11 - Inst Fetch Data Flow



For the Inst Decode & Reg Fetch stage, there are just two logic blocks which nearly all the data flows include. Creating the core data flow was therefore just a case of determining how many pipeline registers to use. To do this, the JAL, BRANCH and LOAD data flows were superimposed on top of each other as together they contain every pipeline register used in the decode stage.

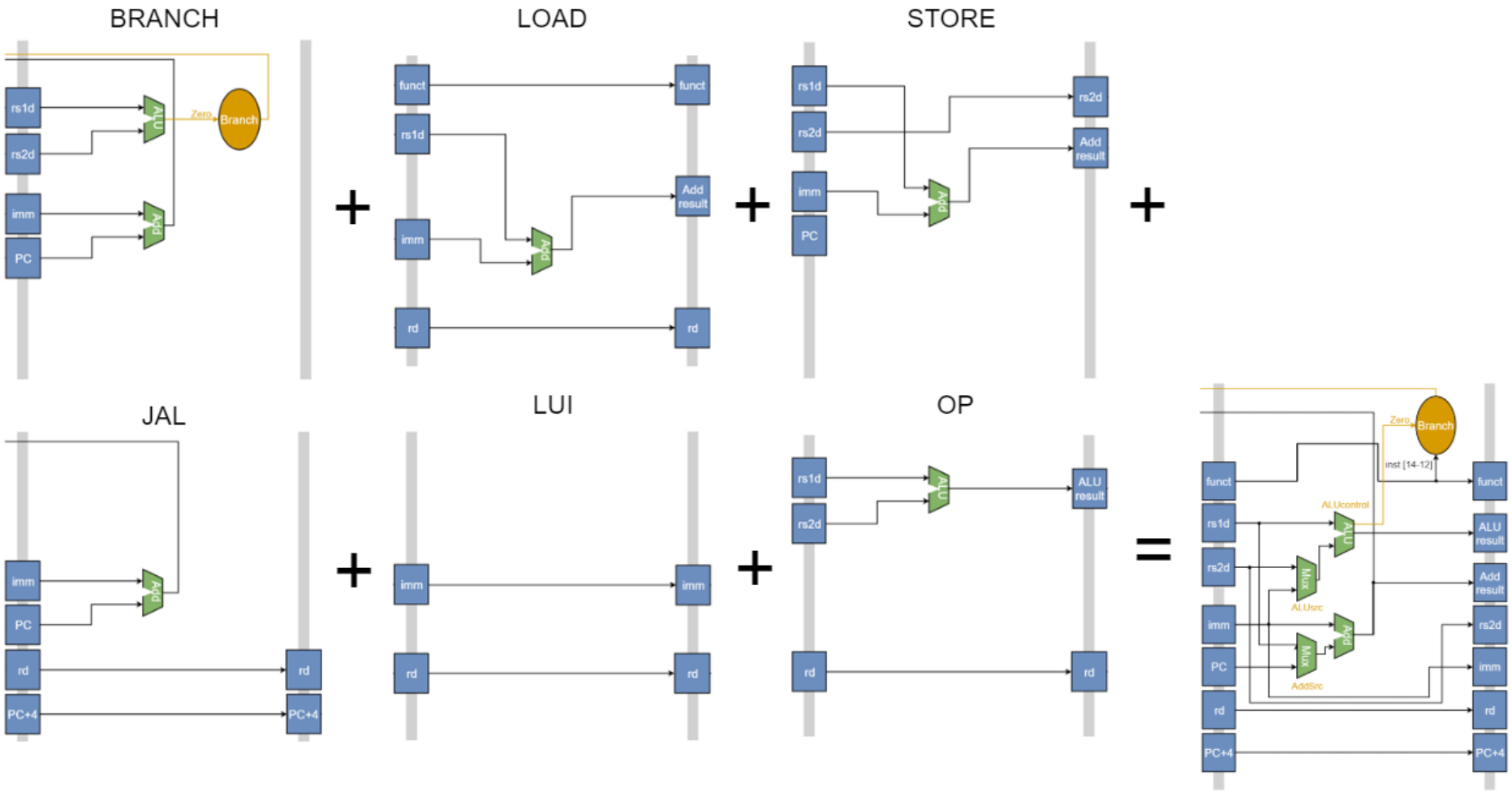
Figure 3.2:12 – Inst Decode & Reg Fetch Data Flow



No pipeline registers can be combined without increasing the critical path.

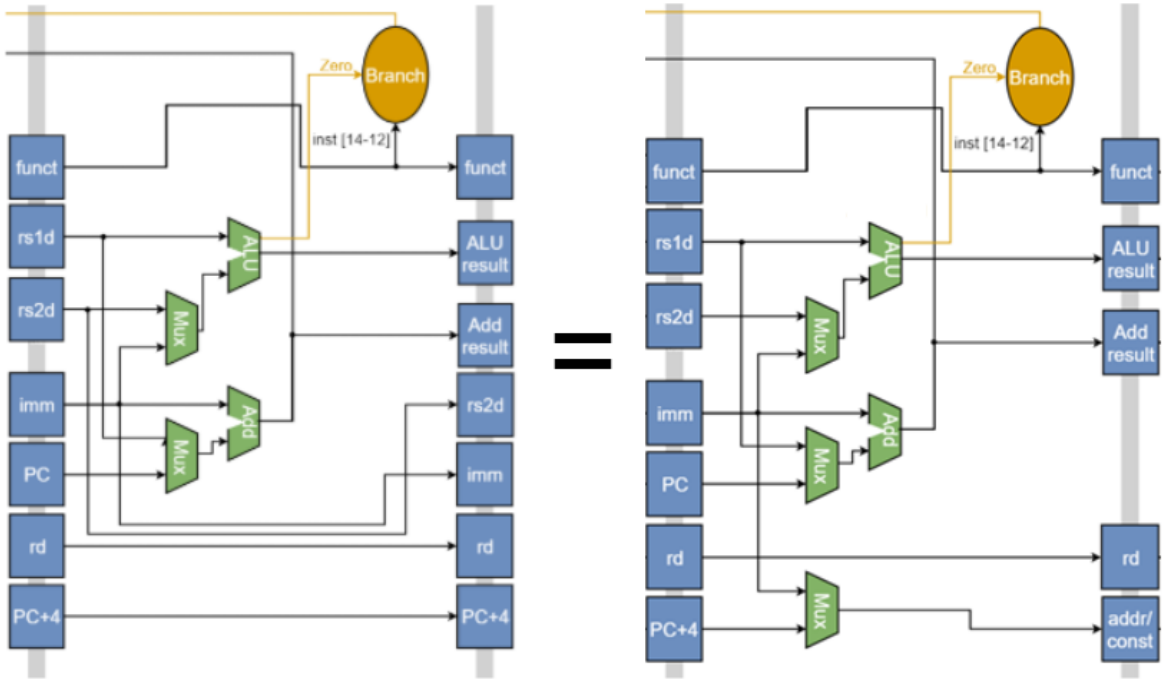
For the Inst Execute stage, there are three logic blocks and the BRANCH data flow uses all of them, so it was used as the base for the core data flow. Any other opcode data flows that used additional pipeline registers were superimposed on top of the BRANCH data flow to create the core data flow.

Figure 3.2:13 - Execute Data Flow



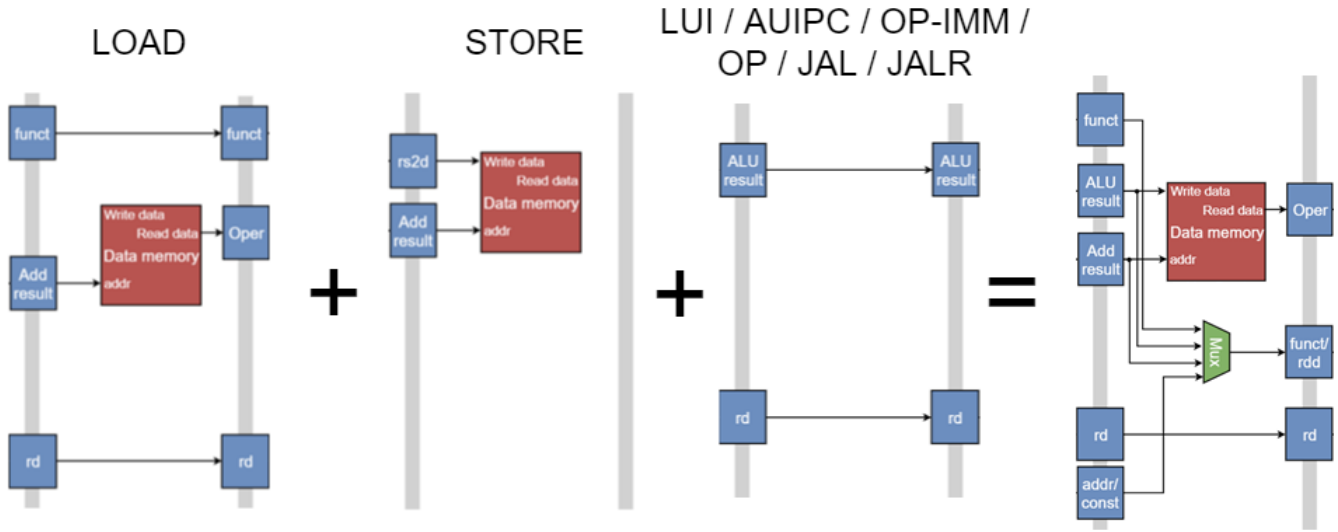
The **imm** and **PC+4** pipeline registers can be combined using a multiplexer, without increasing the critical path. The **rs2d** and **ALU result** registers can also be combined, using the ALU as a passthrough.

Figure 3.2:14 – Simplified Execute Data Flow



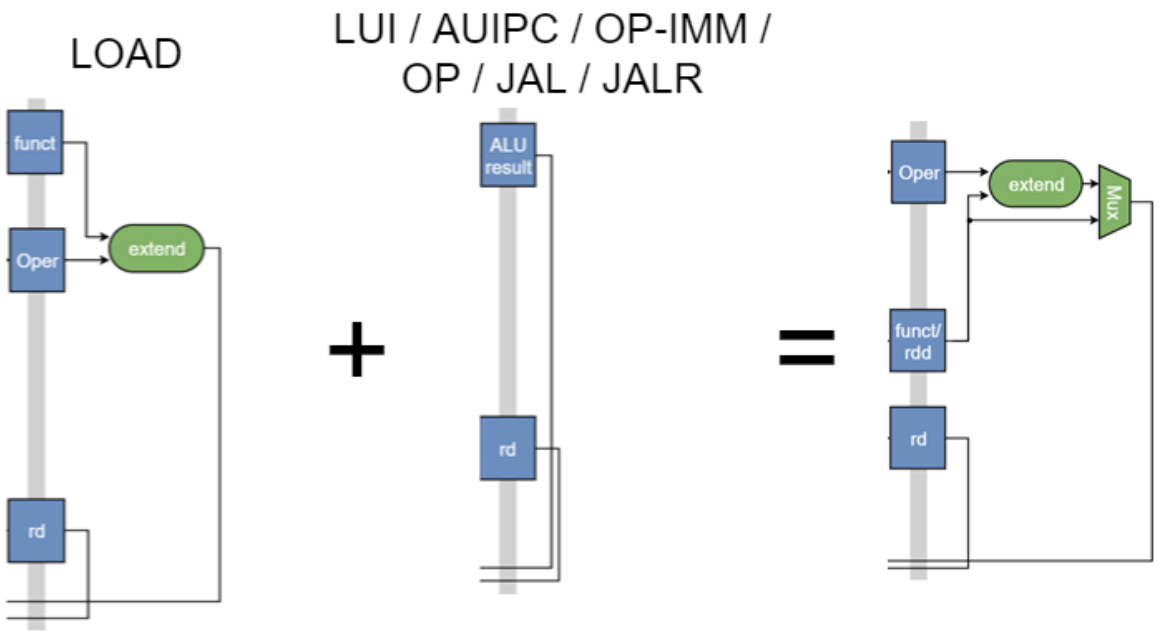
Only the STORE and LOAD data flows use any logic blocks in the Mem Access stage, so they were superimposed on top of each other. All other opcode data flows simply passed through the Mem Access stage, so were combined into a single pipeline register using a multiplexer.

Figure 3.2:15 - Mem Access Data Flow



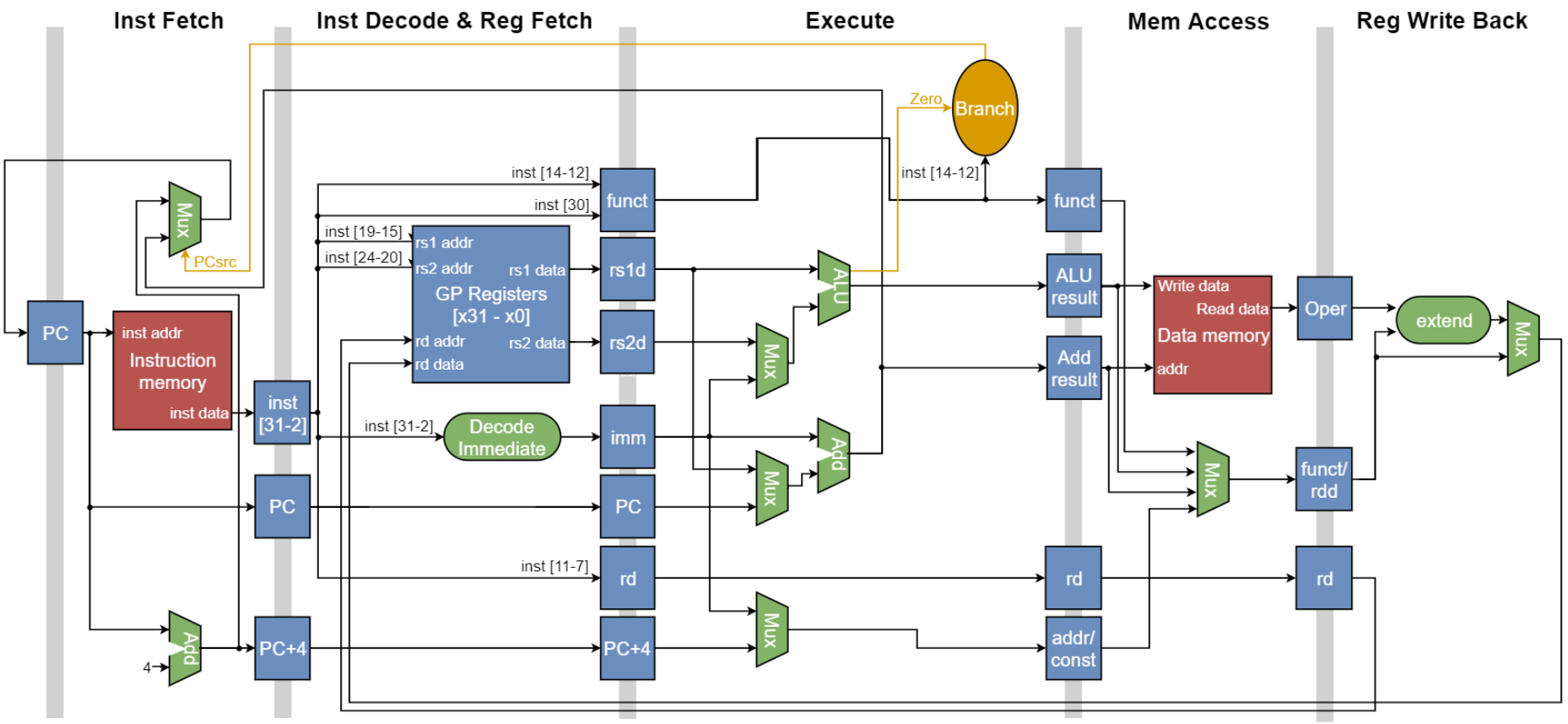
Finally, the LOAD data flow is the only one that uses any logic blocks in for the Write Back stage, so it was used as the base for the core data flow. All other opcode data flows simply passed through the Write Back stage to write straight into a general-purpose register.

Figure 3.2:16 - Write Back Data Flow



Combining all the core data flow stages together gives the complete data flow for the entire core *[figure 3.2:17]*.

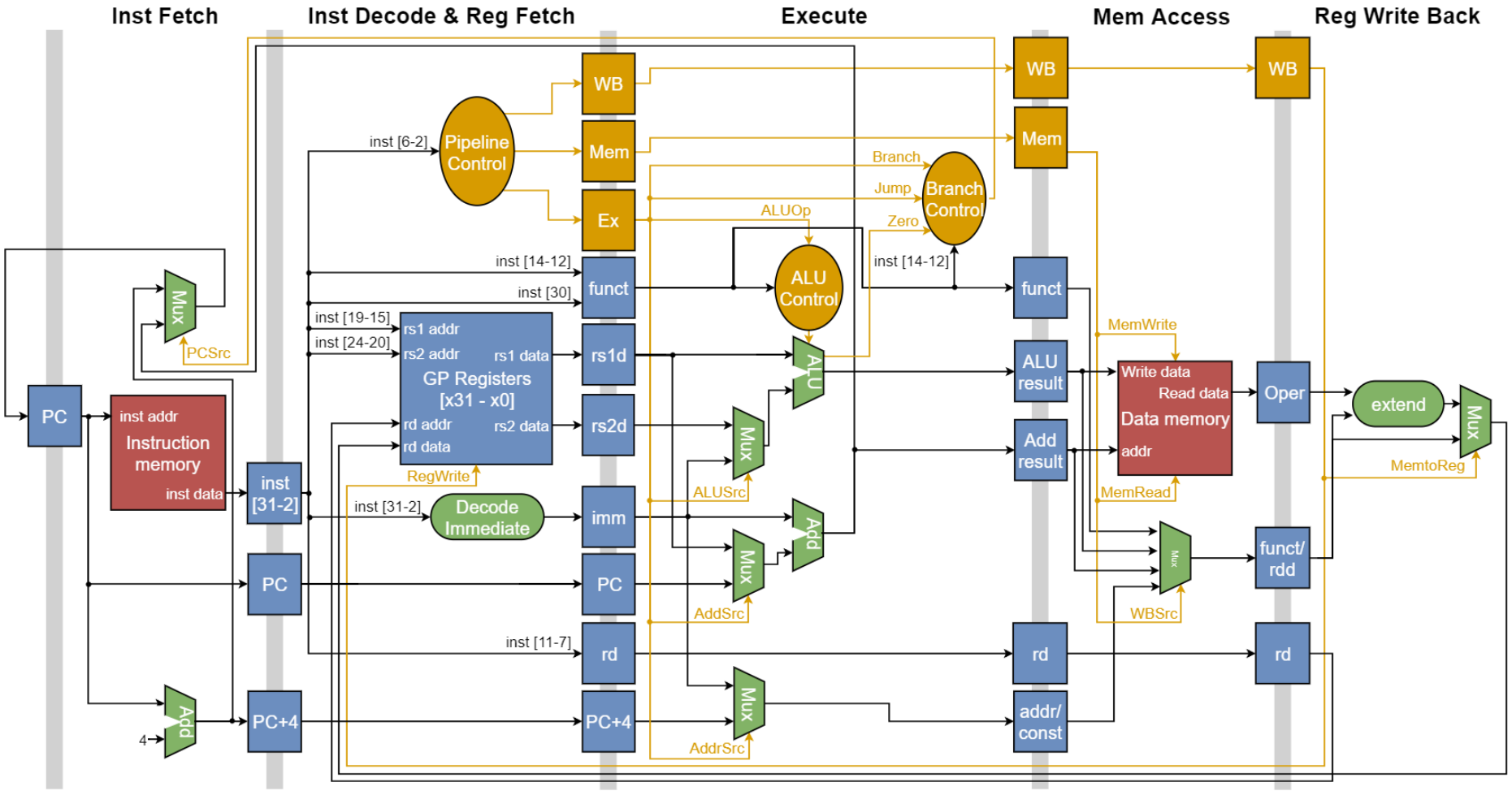
Figure 3.2:17 - Core Data Flow



3.3 Control

To control the flow of data through the core based on the opcode, the core requires control logic *[figure 3.3:1]*. There are three control blocks, the first detailed in section 3.3.1 controls the flow of data through the pipeline. The second, detailed in section 3.3.2 controls the arithmetic/logic operation that will be performed in the ALU. The final control block, detailed in section 3.3.3 controls the source for the program counter (PC+4 or jump/branch).

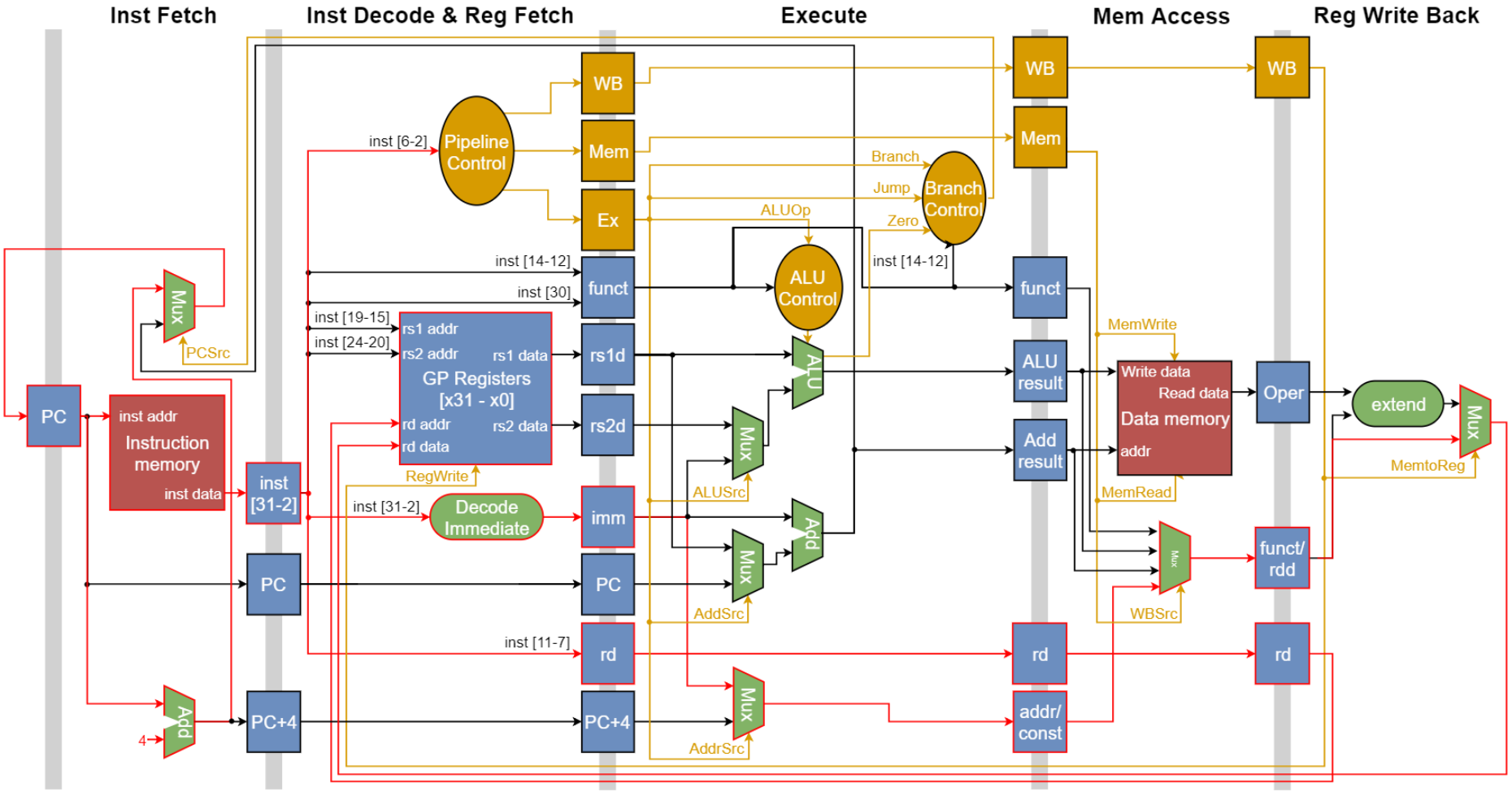
Figure 3.3:1 - Core Control & Data Flow



3.3.1 Pipeline Control

The Pipeline Control block takes in the opcode and decodes it into the appropriate control signals. To build up the truth table for the pipeline control logic, the path of data for each individual opcode was traced out through the core's full data flow. LUI is used as an example *[figure 3.3:2]* and all nine can be found in Appendix A.

Figure 3.3:2 - LUI Control & Data Flow



After tracing the path for each opcode, the truth table can be drawn *[table 3.3:1]* where “1” represents high, “0” low and “X” don’t care.

Table 3.3:1 - Control Truth Table

Opcode	Control												
	WB		Mem				Ex						
	MemtoReg	RegWrite	MemRead	MemWrite	WBSrc[1]	WBSrc[0]	AddrSrc	AddSrc	ALUSrc	ALUOp[1]	ALUOp[0]	Branch	Jump
LUI	0	1	0	0	0	0	1	X	X	X	X	0	0
AUIPC	0	1	0	0	0	1	X	0	X	X	X	0	0
OP-IMM	0	1	0	0	1	0	X	X	0	0	0	0	0
OP	0	1	0	0	1	0	X	X	1	0	1	0	0
JAL	0	1	0	0	0	0	0	0	X	X	X	0	1
JALR	0	1	0	0	0	0	0	1	X	X	X	0	1
BRANCH	X	0	0	0	X	X	X	0	1	1	0	1	0
LOAD	1	1	1	0	1	1	X	1	X	X	X	0	0
STORE	X	0	0	1	X	X	X	1	1	1	1	0	0

Don’t care conditions can be treated as either a high or low condition. As such, it is likely that the truth table can be compressed. To do this, each column of the truth table was compared to identify matching control signals *[table 3.3:2]*. Any matches were colour coded, and once every match was found, they were paired up.

Table 3.3:2 – Matching Pipeline Control Signals

Opcode	Control												
	WB		Mem				Ex						
	MemtoReg	RegWrite	MemRead	MemWrite	WBSrc[1]	WBSrc[0]	AddrSrc	AddSrc	ALUSrc	ALUOp[1]	ALUOp[0]	Branch	Jump
LUI	0	1	0	0	0	0	1	X	X	XXX	XX	0	0
AUIPC	0	1	0	0	0	1	X	0	X	XXX	XX	0	0
OP-IMM	0	1	0	0	1	0	X	X	0	000	00	0	0
OP	0	1	0	0	1	0	X	X	1	000	11	0	0
JAL	0	1	0	0	0	0	0	0	X	XXX	XX	0	1
JALR	0	1	0	0	0	0	0	1	X	XXX	XX	0	1
BRANCH	X	0	0	0	X	X	X	0	1	111	00	1	0
LOAD	1	1	1	0	1	1	X	1	X	XXX	XX	0	0
STORE	X	0	0	1	X	X	X	1	1	111	11	0	0

Once paired up, the truth table can be redrawn *[table 3.3:3]*, giving a total of 9 bits instead of 13 bits. This reduction in bits will simplify the control logic and reduce the size of the register needed to propagate the control signals through the pipeline.

Table 3.3:3 - Compressed Control Truth Table

Opcode	Control								
	RegWrite	MemtoReg & MemRead	MemWrite	WBSrc[1]	WBSrc[0] & ALUOp[1]	AddrSrc & ALUSrc	AddSrc & ALUOp[0]	Branch	Jump
LUI	1	0	0	0	0	1	X	0	0
AUIPC	1	0	0	0	1	X	0	0	0
OP-IMM	1	0	0	1	0	0	0	0	0
OP	1	0	0	1	0	1	1	0	0
JAL	1	0	0	0	0	0	0	0	1
JALR	1	0	0	0	0	0	1	0	1
BRANCH	0	0	0	X	1	1	0	1	0
LOAD	1	1	0	1	1	X	1	0	0
STORE	0	0	1	X	1	1	1	0	0

3.3.2 ALU Control

The ALU needs to perform different functions on the operands fed into it based on an instruction's opcode and function. The ALU Control block takes in the ALUOp (decoded by pipeline Control block), funct 3 (inst[14-12]) and funct 7 (inst[31-25]). In the base-integer instruction set, only a single bit of funct 7 changes (inst[30]) so the other bits can be ignored. First, each ALUOp and funct combination was assigned the appropriate ALU function *[table 3.3:6]*.

Table 3.3:4 - ALU Functions

Key
Integer Computation
Control Transfer
Load/Store

opcode	ALUOp	inst	funct (inst[30] [14-12])	ALU Function
BRANCH	10	BEQ	X 000	SUB
		BNE	X 001	
		BLT	X 100	SLT
		BGE	X 101	
		BLTU	X 110	SLTU
		BGEU	X 111	
STORE	11	SB	X 000	PASS
		SH	X 001	
		SW	X 010	
OP-IMM	00	ADDI	X 000	ADD
		SLTI	X 010	SLT
		SLTIU	X 011	SLTU
		XORI	X 100	XOR
		ORI	X110	OR
		ANDI	X 111	AND
		SLLI	0 001	SLL
		SRLI	0 101	SRL
		SRAI	1 101	SRA
OP	01	ADD	0 000	ADD
		SUB	1 000	SUB
		SLL	0 001	SLL
		SLT	0 010	SLT
		SLTU	0 011	SLTU
		XOR	0 100	XOR
		SRL	0 101	SRL
		SRA	1 101	SRA
		OR	0 110	OR
		AND	0 111	AND

Each ALU function was then assigned a binary value *[table 3.3:5],* so that it could be implemented in logic.

Table 3.3:5 - ALU Operations

ALU Function	Description	ALU Control
ADD	output = ±a + ±b	0000
SLT	if (±a < ±b) then output = 1	0001
SLTU	if (a < b) then output = 1	0010
AND	output = a AND b	0011
OR	output = a OR b	0100
XOR	output = a XOR b	0101
SLL	output = a << b (logical left shift)	0110
SRL	output = a >> b (logical right shift)	0111
SRA	output = a >> b (arithmetic right shift)	1000
SUB	output = ±a – ±b	1001
PASS	output = b	1010

3.3.3 Branch Control

The Branch Control block controls the source for the program counter (PCSrc). If PCSrc is set low, the program counter will get PC+4. If PCSrc is set high, the program counter will get the result from the Add block in the execute stage. The PCSrc should be set high when there is a jump instruction or a branch instruction whose condition is met *[table 3.3:6]*. For example, PCSrc is set high if there is a BEQ (branch if equal) instruction and the result from the ALU is zero (two operands are equal). If the instruction is not a jump or branch, PCSrc should be set low.

Table 3.3:6 - Branch Control

Zero	Jump	Branch	funct3 (inst[14-12])	PCSrc
X	1	X	X	1
1	0	1	BEQ (000)	1
0	0	1	BNE (001)	1
0	0	1	BLT (100)	1
1	0	1	BGE (101)	1
0	0	1	BLTU (110)	1
1	0	1	BGEU (111)	1

3.4 Memory

As the focus of this project is the microprocessor core itself and not the cache or memory architecture, all memory will simply be implemented as registers. This would be similar to having a complex instruction cache that never misses, meaning every instruction can be fetched in a single clock cycle. These registers will be set up as a pure Harvard architecture. Both will be Byte addressable and the data memory will support misaligned access. This will give students a good idea of how a microprocessor core interfaces with cache without having to design and implement a full caching hierarchy.

4 Implementation

This section will cover the FPGA implementation of the design detailed in section 3. The language chosen was VHDL as it is the most popular hardware description language in Europe. VHDL is, therefore, very likely to be covered on most computer & architecture courses in Europe. Every line of code will not be discussed, instead, this section will focus on each pipeline stage and key implementation choices that were made. Small code snippets will be used as examples, but the VHDL code in its entirety can be found in Appendix B.

4.1 Custom Packages

One of the first key optimisations made was the creation of a package which defines shared constants and data types used throughout the implementation. Defining constants and data types that are repeatedly used, in a package *[figure 4.1:1]* and giving them meaningful names will help to make the code easier to read. Being an educational tool, the readability of the VHDL code is arguably the most important quality that it must have.

Figure 4.1:1 - U32_types.vhd

```
5   package u32_types is
6       -- set bit length of architecture
7       constant xlen           : natural := 31;
8
9       -- opcode suffix, always 11 for RV32I architectures
10      constant suffix : std_logic_vector(1 downto 0) := "11";
11
12      -- size of ram in bytes (should be a power of 2 e.g. 256, 512, 1024, etc)
13      constant data_mem_size : natural := 256;
14      constant inst_mem_size : natural := 256;
15
16      -- subtypes used throughout the u32 core
17      subtype word_vector is std_logic_vector(31 downto 0);
18      subtype half_word_vector is std_logic_vector(15 downto 0);
19      subtype byte_vector is std_logic_vector(7 downto 0);
20      subtype nibble_vector is std_logic_vector(3 downto 0);
21      subtype inst_vector is std_logic_vector(31 downto 2);
22      subtype opcode_vector is std_logic_vector(6 downto 2);
23      subtype addr_vector is std_logic_vector(4 downto 0);
24
25      -- opcodes reduced down from 7 bits to 5 by excluding the architectures suffix
26      constant lui    : opcode_vector := "01101";
27      constant auipc  : opcode_vector := "00101";
28      constant opimm  : opcode_vector := "00100";
29      constant op     : opcode_vector := "01100";
30      constant jal    : opcode_vector := "11011";
31      constant jalr   : opcode_vector := "11001";
32      constant branch : opcode_vector := "11000";
33      constant load   : opcode_vector := "00000";
34      constant store  : opcode_vector := "01000";
35
```

```
36      -- alu cotnrol
37      constant alu_add    : nibble_vector := "0000";
38      constant alu_slt    : nibble_vector := "0001";
39      constant alu_sltu   : nibble_vector := "0010";
40      constant alu_and    : nibble_vector := "0011";
41      constant alu_or     : nibble_vector := "0100";
42      constant alu_xor    : nibble_vector := "0101";
43      constant alu_sll    : nibble_vector := "0110";
44      constant alu_srl    : nibble_vector := "0111";
45      constant alu_sra    : nibble_vector := "1000";
46      constant alu_sub    : nibble_vector := "1001";
47      constant alu_pass   : nibble_vector := "1010";
48
49      function log2 (x : natural) return natural;
50  end package u32_types;
```

4.2 Instruction Fetch

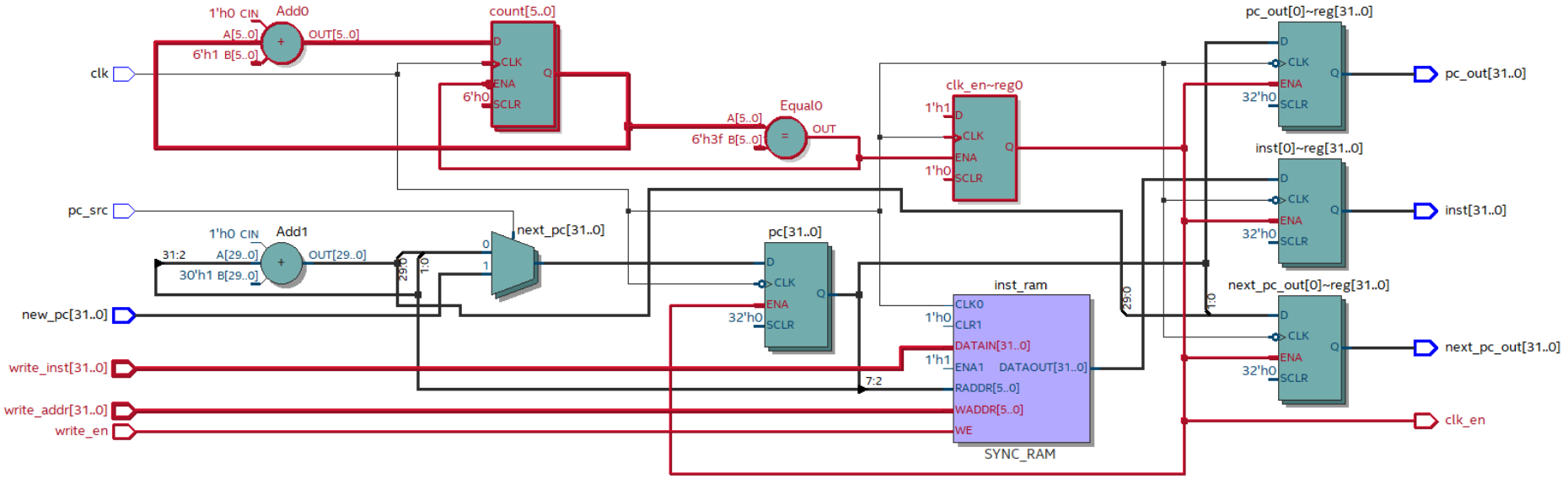
The Inst Fetch pipeline stage is one of the simplest as the logic is fully described within a single entity. The instruction memory is inferred as BRAM instead of pure logic elements. Logic elements are general-purpose and could be used to create memory elements in a FPGA. However, BRAM is specially designed to be used for memory elements. It is, therefore, best practice to use BRAM were possible as it increases performance and utilises less area on the FPGA. In this case, the BRAM was inferred as simple dual-port synchronous RAM with old data read-during-write behaviour [8]. This was done to allow new instructions to be written into instruction memory while the core is running. The size of the instruction memory is defined in the u32_types package, in this case, it was set to 256 bytes so it could store 64 instructions. This should be more than enough for testing purposes. The lines of code which infer the BRAM have been highlighted in blue *[figure 4.2:1]*.

Additional logic not present in the design has been added for testing purposes. This additional logic has been highlighted in red in the VHDL code *[figure 4.2:1]* and on the schematic *[figure 4.2:2]*. It allows a testbench to fill the instruction memory by stalling the pipeline on boot. That would be 64 clock cycles in this case.

Figure 4.2:1 - U32_Inst_Fetch.vhd

```
20      type ram is array (0 to inst_mem_xlen) of word_vector;
21
22      signal next_pc, pc, inst_reg            : word_vector   := (others => '0');
23      signal inst_mem                         : ram           := (others => (others => '0'));
24      signal true_write_addr, true_read_addr  : natural       := 0;
25  begin
26      -- concurrent statements
27      -- convert byte address into word address
28      true_read_addr <= to_integer(unsigned(pc)) / 4;
29      true_write_addr <= to_integer(unsigned(write_addr)) / 4;
30
31      -- read instruction out of instruction memory
32      inst_reg <= inst_mem(true_read_addr);
33
34      -- update program counter
35      next_pc <=    new_pc when pc_src = '1' else
36                    pc + increment;
37
38      -- sequential statements
39      process (clk)
40          variable count  : natural range 0 to inst_mem_xlen  := 0;
41      begin
42          if rising_edge(clk) then
43              -- stall pipeline until instruction memory filled
44              if count = inst_mem_xlen then
45                  clk_en <= '1';
46              else
47                  count := count + 1;
48              end if;
49
50              -- write instruction into instruction memory
51              if write_en = '1' then
52                  inst_mem(true_write_addr) <= write_inst;
53              end if;
54          elsif falling_edge(clk) then
55              -- pipeline registers
56              if clk_en = '1' then
57                  pc <= next_pc;
58                  pc_out <= pc;
59                  next_pc_out <= pc + increment;
60                  inst <= inst_reg;
61              end if;
62          end if;
63      end process;
```

Figure 4.2:2 - Inst Fetch Schematic



4.3 Decode

To make the code for the decode stage more readable, the immediate decoder, pipeline controller, and general-purpose registers were defined as sperate entities. The immediate decoder uses selected signal assignment (with/select) to decode the immediates as detailed in section 3.1.2 *[figure 4.3:1]*. Selected signal assignment was used over a case statement, as it makes it more apparent to the reader that the immediate decoder is strictly combinational logic.

Figure 4.3:1 - U32_Immediate_Decoder

```
19      -- decode immediate
20      with opcode select
21      imm <=  inst(xlen downto 12) & (11 downto 0 => '0')                                          when lui | auipc, -- U-type
22              (xlen downto 20 => inst(xlen)) & inst(19 downto 12) & inst(20) & inst(30 downto 21) & '0'   when jal, -- J-type
23              (xlen downto 12 => inst(xlen)) & inst(7) & inst(30 downto 25) & inst(11 downto 8) & '0'     when branch, -- B-type
24              (xlen downto 11 => inst(xlen)) & inst(30 downto 25) & inst(11 downto 7)                     when store, -- S-type
25              (xlen downto 11 => inst(xlen)) & inst(30 downto 20)                                         when others; -- I-type
```

The pipeline controller also uses selected signal assignment to decode the control signals as detailed in section 3.3.1 *[figure 4.3:2]*. Again, selected signal assignment was used to make it apparent that the pipeline controller is strictly combinational logic.

Figure 4.3:2 - U32_Controller.vhd

```
17      -- decode control signals
18      with opcode select
19      control <=  "100001000" when lui,
20                  "100010000" when auipc,
21                  "100101100" when op,
22                  "100000001" when jal,
23                  "100000101" when jalr,
24                  "000011010" when branch,
25                  "110110100" when load,
26                  "001011100" when store,
27                  "100100000" when others; -- opimm
```

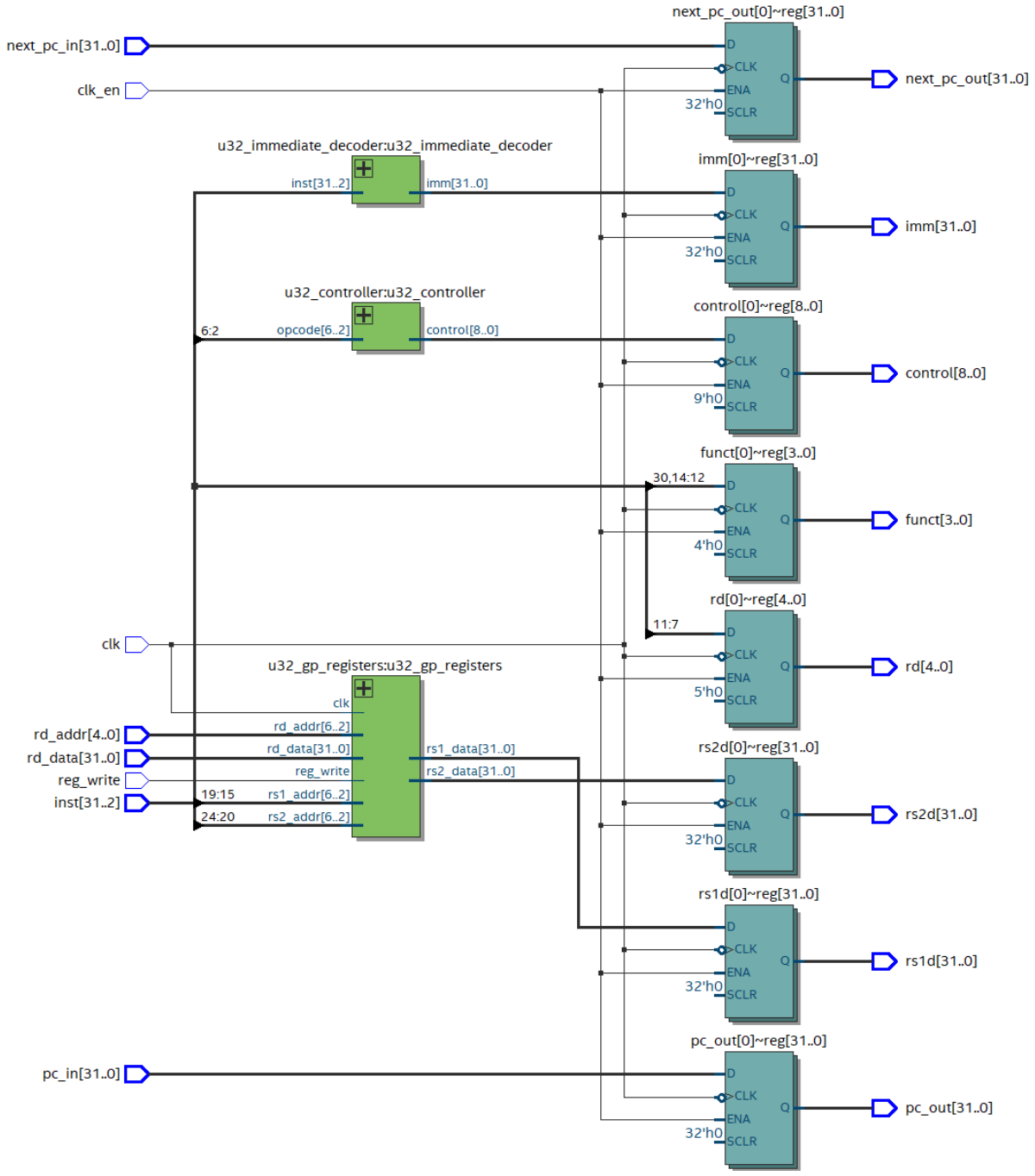
The general-purpose registers are inferred as BRAM *[figure 4.3:3]* for the same reason as the instruction memory discussed in section 4.2. In this case, the BRAM was inferred as true dual-port synchronous RAM [8]. This was done to allow two reads and one write in a single clock cycle. This behaviour is required for cases where an instruction reads from two source registers in the decode stage, while a previous instruction writes back to a destination register in the writeback stage.

Figure 4.3:3 - U32_GP_Registers

```
17      type word_matrix is array (0 to 31) of word_vector;
18      signal gp_registers : word_matrix := (others => (others => '0'));
19  begin
20      -- concurrent statements (read)
21      rs1_data <= gp_registers(to_integer(unsigned(rs1_addr)));
22      rs2_data <= gp_registers(to_integer(unsigned(rs2_addr)));
23      -- sequential statements (write)
24      process begin
25          wait until rising_edge(clk);
26          if (reg_write = '1' and rd_addr /= "00000") then
27              gp_registers(to_integer(unsigned(rd_addr))) <= rd_data;
28          end if;
29      end process;
```

The immediate decoder, pipeline controller, and general-purpose registers were instantiated, and pipeline registers added to create the decode stage. Having done that, the schematic *[figure 4.3:4]* perfectly matches the decode data flow detailed in section 3.2.3.

Figure 4.3:4 - Decode Schematic



4.4 Execute

To make the code for the execute stage more readable, the ALU, ALU controller and branch controller were defined as sperate entities. The ALU uses a case statement to execute the integer and arithmetic operations as detailed in section 3.3.2 *[figure 4.4:1]*. The case statement was used over selected or conditional signal assignment (when/else), as the SLT and SLTU operations check multiple conditions (if within an if). This cannot be done with selected or conditional signal assignment.

Figure 4.4:1 - U32_ALU.vhd

```
25      shamt <= operand2(4 downto 0);
26      result <= aluresult;
27      zero <= '1' when aluresult = (xlen downto 0 => '0') else '0'; -- set zero flag high when result is 0
28
29      process (alu_control, operand1, operand2, shamt)
30      begin
31          -- perform the approiate arithmatic/logic function based on ALU control signal
32          case alu_control is
33              when alu_slt =>
34                  if operand1 < operand2 then
35                      aluresult <= (xlen downto 1 => '0') & '1';
36                  else
37                      aluresult <= (others => '0');
38                  end if;
39              when alu_sltu =>
40                  if unsigned(operand1) < unsigned(operand2) then
41                      aluresult <= (xlen downto 1 => '0') & '1';
42                  else
43                      aluresult <= (others => '0');
44                  end if;
45              when alu_and =>
46                  aluresult <= operand1 and operand2;
47              when alu_or =>
48                  aluresult <= operand1 or operand2;
49              when alu_xor =>
50                  aluresult <= operand1 xor operand2;
51              when alu_sll =>
52                  aluresult <= std_logic_vector(shift_left(unsigned(operand1), to_integer(unsigned(shamt))));
53              when alu_srl =>
54                  aluresult <= std_logic_vector(shift_right(unsigned(operand1), to_integer(unsigned(shamt))));
55              when alu_sra =>
56                  aluresult <= std_logic_vector(shift_right(signed(operand1), to_integer(unsigned(shamt))));
57              when alu_sub =>
58                  aluresult <= operand1 - operand2;
59              when alu_pass =>
60                  aluresult <= operand2;
61              when others =>
62                  aluresult <= operand1 + operand2;
63          end case;
64      end process;
```

The ALU controller uses selected signal assignment to decode the ALU control signals as detailed in section 3.3.2 *[figure 4.4:2]*. Selected signal assignment was used to make it apparent that the ALU controller is strictly combinational logic. As many instructions that use the ALU do not contain funct7 as detailed in section 3.1.4, the control signal often does not care whether bit inst[30] is a 1 or 0 *[table 3.3:4]*. While IEEE std_logic data types do support don't care conditions, they are simply ignored by the synthesiser. To get around this, two variants of these constants were defined. One with inst[30] set to 1 and one with it set to 0.

Figure 4.4:2 - U32_ALU_Controller.vhd

```
78      inst_alu <= aluop & funct; -- cobined aluop and funct so with/select can be used
79
80      -- decode alu control signal
81      with inst_alu select
82      alu_control <=  alu_slt     when blt_inst1  | bge_inst1  | slti_inst1 | blt_inst0  | bge_inst0  | slti_inst0  | slt_inst,
83                      alu_sltu    when bltu_inst1 | bgeu_inst1 | sltiu_inst1 | bltu_inst0 | bgeu_inst0 | sltiu_inst0 | sltu_inst,
84                      alu_and     when andi_inst1 | andi_inst0 | and_inst,
85                      alu_or      when ori_inst1  | ori_inst0 | or_inst,
86                      alu_xor     when xori_inst1 | xori_inst0 | xor_inst,
87                      alu_sll     when slli_inst | sll_inst,
88                      alu_srl     when srli_inst | srl_inst,
89                      alu_sra     when srai_inst | sra_inst,
90                      alu_sub     when sub_inst | beq_inst0 | beq_inst1 | bne_inst0 | bne_inst1,
91                      alu_pass    when sb_inst1 | sh_inst1 | sw_inst1 | sb_inst0 | sh_inst0 | sw_inst0,
92                      alu_add     when others;
```

The Branch controller uses conditional signal assignment to control the program counter source as detailed in section 3.3.3 *[figure 4.4:3]*. Conditional signal assignment (when/else) was used as when jump is 1, the value of branch, zero & funct doesn't matter. If selective signal assignment was used, every possible combination of branch, zero & funct when jump is 1 would need to be defined.

Figure 4.4:3 - U32_Branch_controller.vhd

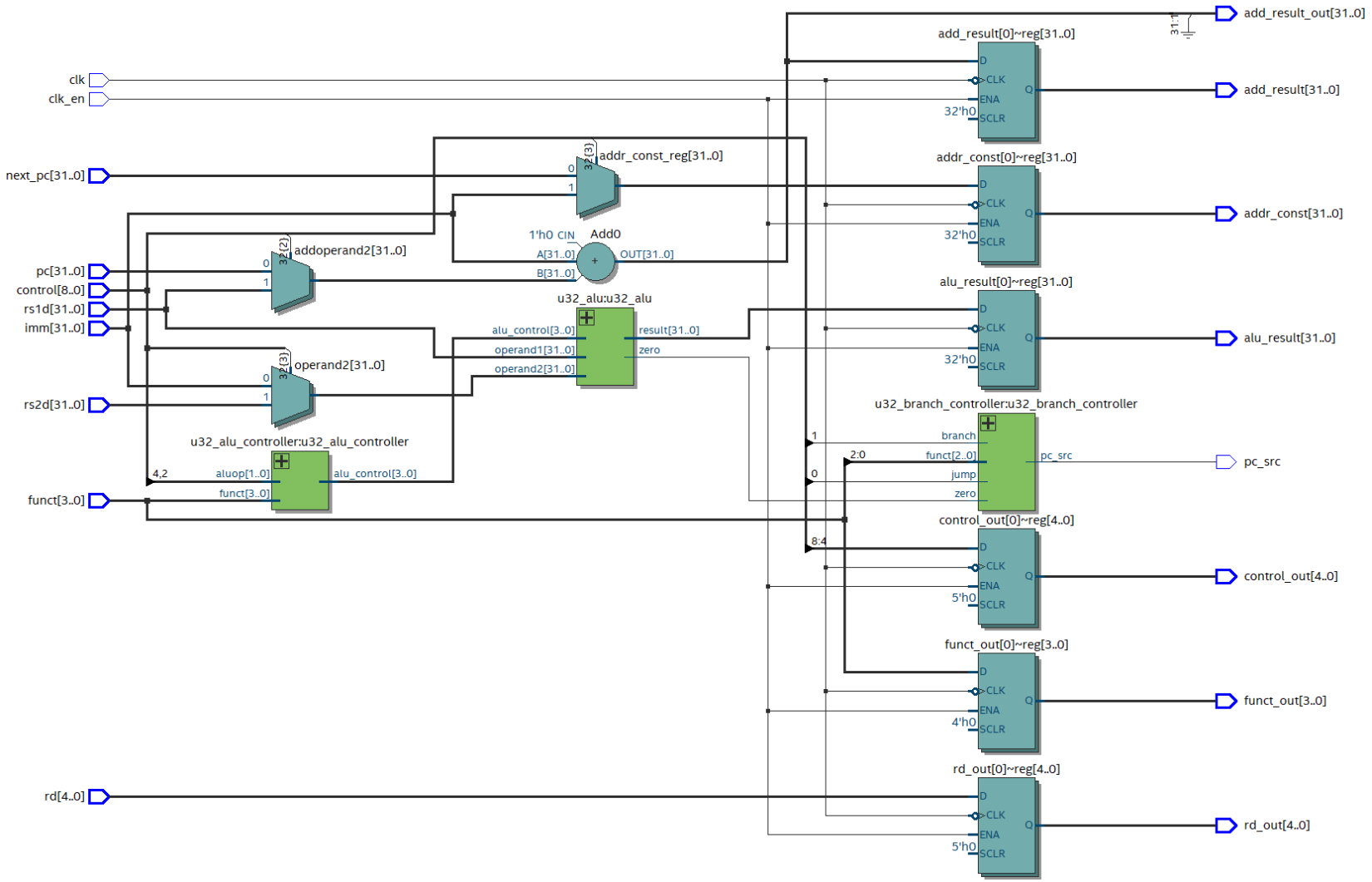
```
18      -- set pc_src high when jump or branch + condition met
19      pc_src <=   '1' when jump = '1' else
20                  '1' when branch & zero & funct = "11000" else -- BEQ and equal
21                  '1' when branch & zero & funct = "10001" else -- BNE and not equal
22                  '1' when branch & zero & funct = "10100" else -- BLT and less than
23                  '1' when branch & zero & funct = "11101" else -- BGE and greater than or equal
24                  '1' when branch & zero & funct = "10110" else -- BLTU and less than
25                  '1' when branch & zero & funct = "11111" else -- BGEU and greater than or equal
26                  '0';
```

The ALU, ALU controller and branch controller were instantiated, and pipeline registers added to create the execute stage. The multiplexers were implemented using conditional signal assignment and add logic block as a standard signal assignment *[figure 4.4:4]*. Having done that, the schematic *[figure 4.4:5]* perfectly matches the execute data flow detailed in section 3.2.3.

Figure 4.4:4 - U32_Execute.vhd

```
37      -- connect add_result directly to output pin. Perminalteley set bit 0 to 0 as jump and branch
38      -- should only increment the proghram counter in multiples of two bytes
39      add_result_out <= add_result_reg(xlen downto 1) & '0';
40
41      -- control source of operands into ALU, Add unit and addr/const pipeline register (multiplexers)
42      operand2 <= rs2d when addrsrc_alusrc = '1' else imm;
43      addoperand2 <= rs1d when addsrc_aluop0 = '1' else pc;
44      addr_const_reg <= imm when addrsrc_alusrc = '1' else next_pc;
45
46      -- Add unit
47      add_result_reg <= imm + addoperand2;
```

Figure 4.4:5 - Execute Schematic



4.5 Memory Access

The main component of the Mem Access pipeline stage is the data memory which was defined as a sperate entity to improve the readability of the code. The data memory needed to be able to write specific bytes of a word, without overwriting the value of unwritten bytes (byte-enabled). It must also support misaligned access so a word can be read/written from any byte address. This functionality could not be implemented with a single BRAM so it was instead implemented with logic elements *[figure 4.5:1]*.

Figure 4.5:1 – U32_Data_Memory.vhd

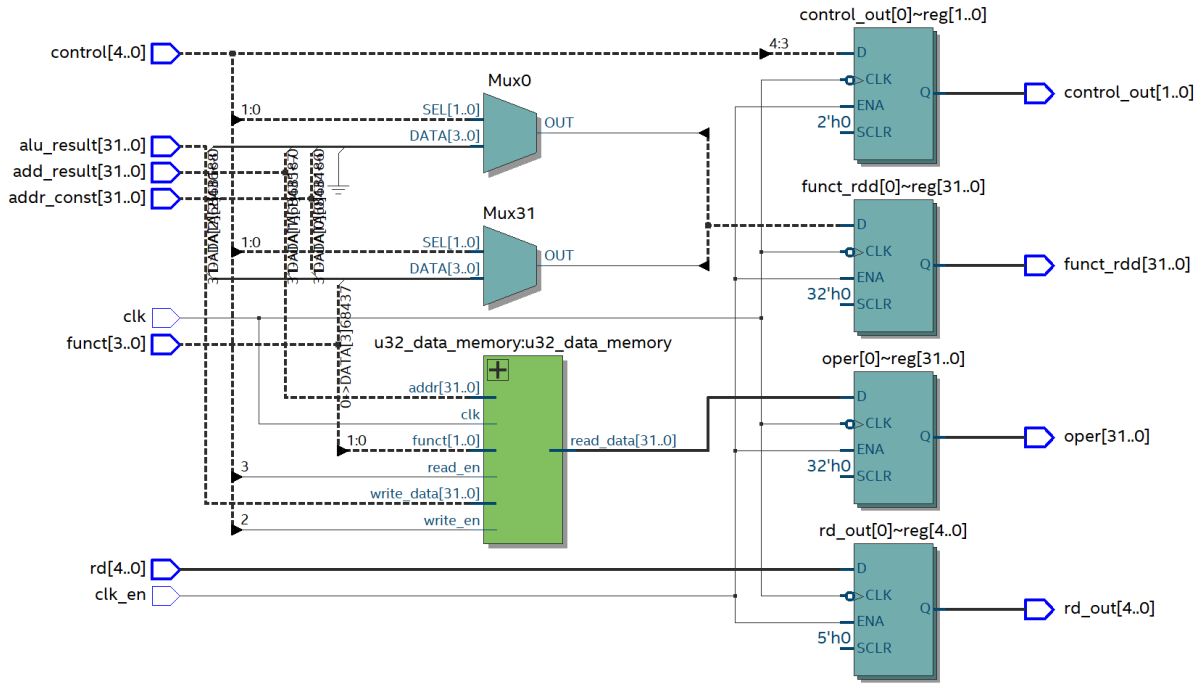
```
35      -- read word from data memory, padding with zeros if the address is the last, second to last or third to
36      -- last byte address
37      data <= x"000000"   & data_mem(ram_addr)       when ram_addr = last_byte else
38              x"0000"     & data_mem(ram_addr + 1)
39                          & data_mem(ram_addr)       when ram_addr = second_last_byte else
40              x"00"       & data_mem(ram_addr + 2)
41                          & data_mem(ram_addr + 1)
42                          & data_mem(ram_addr)       when ram_addr = third_last_byte else
43              data_mem(ram_addr + 3) &
44              data_mem(ram_addr + 2) &
45              data_mem(ram_addr + 1) &
46              data_mem(ram_addr);
47
48      read_data <= data when (read_en = '1') else x"00000000"; -- do not return data unless read enable is high
49
50      process begin
51          wait until rising_edge(clk);
52          -- write word into data memory when address is anything other than the last, second to last or
53          -- third to last byte address
54          if (write_en = '1' and funct = word and ram_addr /= last_byte and ram_addr /= second_last_byte and
55              ram_addr /= third_last_byte) then
56              data_mem(ram_addr) <= write_data(7 downto 0);
57              data_mem(ram_addr + 1) <= write_data(15 downto 8);
58              data_mem(ram_addr + 2) <= write_data(23 downto 16);
59              data_mem(ram_addr + 3) <= write_data(31 downto 24);
60          -- write first three bytes of word into data mem when address is third to last byte address
61          elsif (write_en = '1' and funct = word and ram_addr /= last_byte and ram_addr /= second_last_byte) then
62              data_mem(ram_addr) <= write_data(7 downto 0);
63              data_mem(ram_addr + 1) <= write_data(15 downto 8);
64              data_mem(ram_addr + 2) <= write_data(23 downto 16);
65          -- write half-word into data mem or last two byte of a word when address is second to last byte address
66          elsif (write_en = '1' and funct /= byte and ram_addr /= last_byte) then
67              data_mem(ram_addr) <= write_data(7 downto 0);
68              data_mem(ram_addr + 1) <= write_data(15 downto 8);
69          -- write byte into data memory
70          elsif (write_en = '1') then
71              data_mem(ram_addr) <= write_data(7 downto 0);
72          end if;
73      end process;
```

After instantiating the data memory, adding the multiplexer as a selected signal assignment *[figure 4.5:2]* and pipeline registers, the schematic *[figure 4.5:3]* perfectly matches the mem access data flow detailed in section 3.2.3.

Figure 4.5:2 - U32_Mem_Access.vhd

```
28      -- control source of data into funct/rdd pipeline register (multiplexer)
29      with wbsrc select
30      funct_rdd_reg <=    addr_const when "00",
31                          add_result when "01",
32                          alu_result when "10",
33                          (31 downto 4 => '0') & funct when others;
```

Figure 4.5:3 - Mem Access Schematic



4.6 Write Back

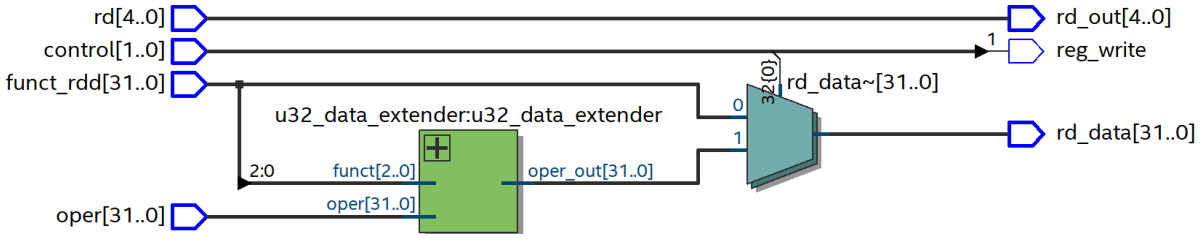
The Write Back pipeline stage is the simplest as the logic could easily be described within a single entity. However, the data extender was defined as its own entity to make the schematic more closely resemble the data flow. Selected signal assignment was used to implement the logic for the data extender to make it apparent that it is fully combinational logic.

Figure 4.6:1 - U32_Data_Extender.vhd

```
19      with funct select
20      oper_out <= (xlen downto 7 => oper(7)) & oper(6 downto 0) when lb, -- sign extend byte
21                  (xlen downto 15 => oper(15)) & oper(14 downto 0) when lh, -- sign extend half word
22                  (xlen downto 8 => '0') & oper(7 downto 0) when lbu, -- zero extend byte
23                  (xlen downto 16 => '0') & oper(15 downto 0) when lhu, -- zero extend half word
24                  oper when others; -- passthrough
```

After instantiating the data extender and adding the multiplexer, the schematic *[figure 4.6:2]* perfectly matches the Writeback data flow detailed in section 3.2.3.

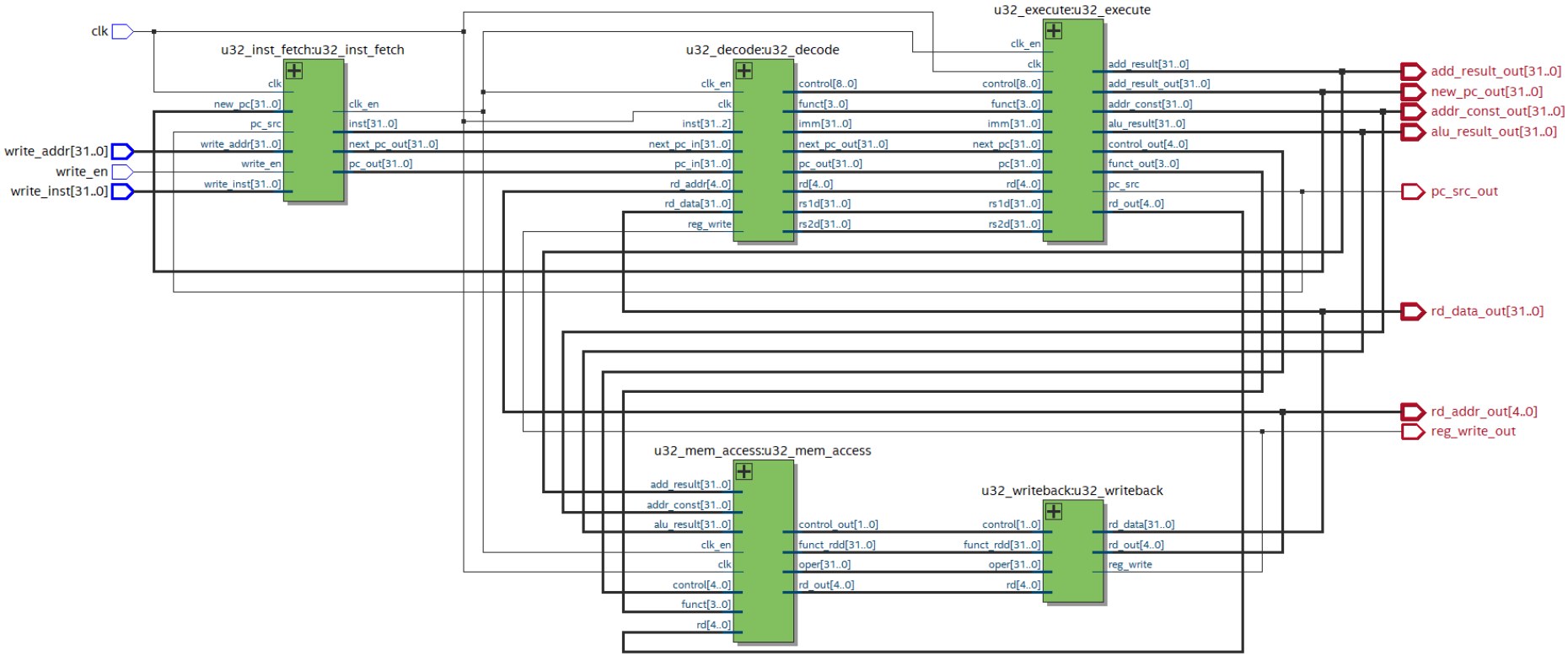
Figure 4.6:2 - Writeback Schematic



4.7 Core

All these pipeline stages were instantiated into a single entity and connected using signals, to form the microprocessor core. The pins highlighted in red on the schematic *[figure 4.7:1]* are used for debugging/testing purposes. These outputs are only taken out of the Execute and Writeback stages. This was done as the Execute stage is where all jump/branches terminate and the Writeback stage is where all other instructions terminate, with the exception of STORE instructions. STORE instruction can be tested with LOAD instructions, which do terminate at the Writeback stage. The write_addr, write_en, and write_inst pins are used to program the microprocessor.

Figure 4.7:1 - Core Schematic



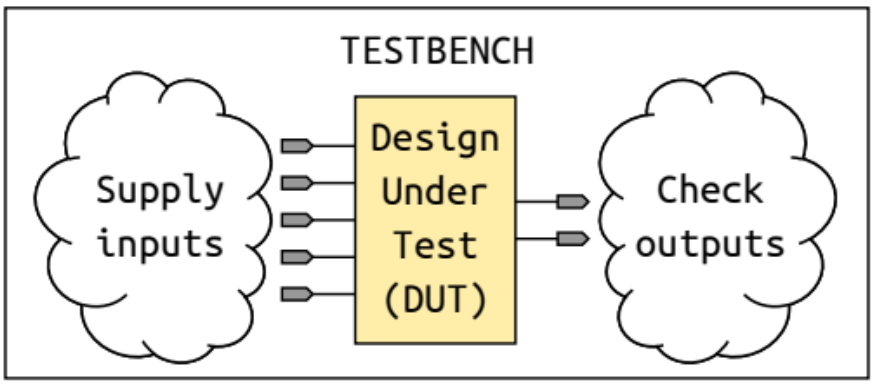
5 Testing Methodology

The approach taken to testing the implementation detailed in section 4 will be covered in this section. Testing was carried out on each VHDL entity, starting with the entities that did not instantiated other entities within them (standalone entities). After each standalone entity had passed testing, the pipeline stages were tested. Finally, after each pipeline stage had passed testing, the core was tested. Testing was carried out in this order as the core would not operate correctly if any single pipeline stage was faulty. Similarly, a pipeline stage would not operate correctly if any of the entities instantiated within that stage were faulty. Therefore, testing in this order (bottom to top) makes the process of isolating and resolving faults much quicker.

5.1 Testbenches

To carry out testing on each entity, custom non-synthesisable VHDL testbenches were created for each entity. These testbenches treat the entity being tested as a black box, supplying test vectors on the entities inputs and checking the outputs against expected results.

Figure 5.1:1 - Test Architecture [9]



This way, even if the architecture of the entity being tested is completely changed, the test will still be valid. Also, the test vectors can easily be changed, added or removed, without having to change the architecture of the testbench.

5.1.1 Standalone Entities & Pipeline Stages

Most of the standalone entities use strictly combinational logic and therefore share a similar architecture for their testbenches. The testbench for the pipeline controller has been used as an example *[figure 5.1:1]* but the code for each testbench can be found in Appendix C. The entities that use this testbench architecture are as follows:

- Pipeline controller
- ALU controller
- ALU
- Data extender
- Immediate Decoder

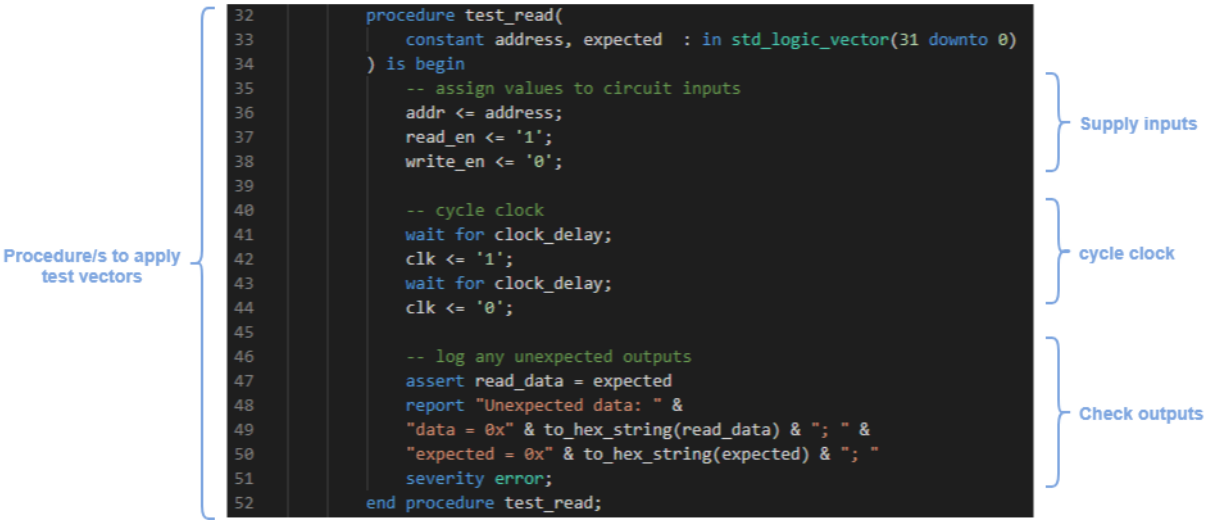
Figure 5.1:2 - Combinational Logic Testbench Architecture


```
15    -- instantiate controller
16    controller : entity work.u32_controller
17    port map (
18        opcode => opcode,
19        control => control
20    );
21
22    process
23        procedure test(
24            constant operation  : in std_logic_vector(4 downto 0);
25            constant expected       : in std_logic_vector(8 downto 0)
26        ) is
27
28        begin
29            -- assign values to circuit inputs
30            opcode <= operation;
31
32            -- wait for controller to respond
33            wait for time_delta;
34
35            -- log any unexpted outputs
36            assert control = expected
37            report "Unexpected result: " &
38            "operation = " & to_string(operation) & "; " &
39            "result = " & to_string(control) & "; " &
40            "expected = " & to_string(expected)
41            severity error;
42        end procedure test;
43    begin
44        test("01101", "100001000"); -- LUI
45        test("00101", "100010000"); -- AUIPC
46        test("11011", "100000001"); -- JAL
47        test("11001", "100000101"); -- JALR
48        test("11000", "000011010"); -- BRANCH
49        test("00000", "110110100"); -- LOAD
50        test("01000", "001011100"); -- STORE
51        test("00100", "100100000"); -- OP-IMM
52        test("01100", "100101100"); -- OP
53        wait;
54    end process;
```

The entities that use sequential logic require a slightly different testbench architecture. The instantiation of the entity to be tested and the definition of test vectors are identical, however, the procedures that apply the test vectors are slightly different. Using the data memory testbench as an example *[figure 5.2:3]*, it is clear that the main difference is the introduction of a clock. The entities that use this testbench architecture are as follows:

- Data Memory
- General-Purpose Registers
- Inst Fetch pipeline stage
- Decode pipeline stage
- Execute pipeline stage
- Mem Access pipeline stage

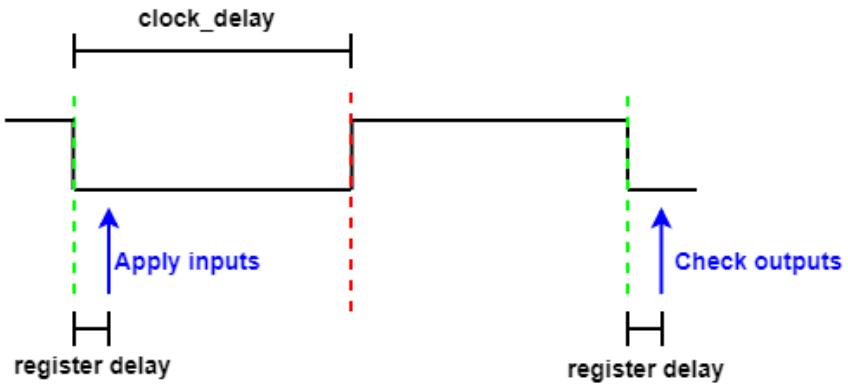
Figure 5.1:3 - Sequential Testbench Architecture




```
32      procedure test_read(
33          constant address, expected  : in std_logic_vector(31 downto 0)
34      ) is begin
35          -- assign values to circuit inputs
36          addr <= address;
37          read_en <= '1';
38          write_en <= '0';
39
40          -- cycle clock
41          wait for clock_delay;
42          clk <= '1';
43          wait for clock_delay;
44          clk <= '0';
45
46          -- log any unexpected outputs
47          assert read_data = expected
48          report "Unexpected data: " &
49          "data = 0x" & to_hex_string(read_data) & "; " &
50          "expected = 0x" & to_hex_string(expected) & "; "
51          severity error;
52      end procedure test_read;
```

In the case of the data memory and general-purpose registers, this does not change the structure of the procedure much. However, how the entity handles the clock, determines when in the clock cycle the outputs should be checked. In the case of the pipeline stages, the testbench has to emulate the delay incurred by the pipeline registers *[figure 5.1:4]*.

Figure 5.1:4 - Testbench Clock



5.1.2 Core

To test the microprocessor core, a VHDL testbench that acted as an assembly compiler and microprocessor programmer was created. It consists of two files, one which acts as the compiler and the other which acts as the programmer. The compiler is simply a custom VHDL package that contains a procedure to produces each instruction *[figure 5.1:5]*, a procedure to load the compiled instructions into the microprocessor's instruction memory *[figure 5.1:6]* and a procedure to cycle the clock enough times to run the program *[figure 5.1:7]*.

Figure 5.1:5 - Example Instruction Compilation (getto_compiler.vhd)

```
362     -- instructions
363     procedure lui(
364         constant rd     : in reg_addr;
365         constant imm    : in u_imm;
366 
367         signal signals  : out std_logic_vector(65 downto 0)
368     ) is
369         constant opcode     : std_logic_vector(6 downto 0)  := "0110111";
370         variable inst       : std_logic_vector(31 downto 0);
371     begin
372         inst := std_logic_vector(to_signed(imm, 20)) & std_logic_vector(to_unsigned(rd, 5)) & opcode;
373         load_inst(inst, signals);
374     end procedure lui;
```

Figure 5.1:6 - Load Instruction (getto_compiler.vhd)

```
320     -- procedure only used with the package iteself
321     procedure load_inst(
322         constant inst     : in std_logic_vector(31 downto 0);
323 
324         signal signals  : out std_logic_vector(65 downto 0)
325     ) is begin
326         signals(64) <= '1'; -- set instruction memory write enable high
327         signals(63 downto 32) <= inst; -- write instruction into instruction memory
328 
329         -- cycle clock
330         wait for clock_delay;
331         signals(65) <= '1';
332         wait for clock_delay;
333         signals(65) <= '0';
334 
335         -- increment write address for instruction memory
336         signals(31 downto 0) <= signals(31 downto 0) + x"00000004";
337     end procedure load_inst;
```

Figure 5.1:7 - Run Program (getto_compiler.vhd)

```
339     procedure run(
340         signal signals  : out std_logic_vector(65 downto 0)
341     ) is begin
342         -- stop writing instruction into isntruction memory
343         signals(64 downto 0) <= (others => '0');
344 
345         -- cycle clock
346         for i in 0 to 69 loop
347             wait for clock_delay;
348             signals(65) <= '1';
349             wait for clock_delay;
350             signals(65) <= '0';
351         end loop;
352         wait;
353     end procedure run;
```

The programmer is the actual testbench that interfaces with the microprocessor core and utilises the producers defined in the compiler. It instantiates the core and maps the inputs to a single signal so they can be easily passed to the compiler *[figure 5.1:8]*.

Figure 5.1:8 - Core Instantiation (Core_testbench.vhd)

```
20      -- instantiate core
21      core : entity work.u32_core
22      port map (
23          -- inputs
24          clk => clk,
25          write_en => write_en,
26          write_inst => write_inst,
27          write_addr => write_addr,
28          -- outputs
29          imm_out => imm,
30          rs1d_out => rs1d,
31          rs2d_out => rs2d,
32          alu_result_out => alu_result,
33          add_result_out => add_result,
34          new_pc_out => new_pc,
35          pc_src_out => pc_src,
36          rd_data_out => rd_data,
37          rd_addr_out => rd_addr,
38          reg_write_out => reg_write
39      );
40
41      -- combine signals to pass to make it easier to pass to the compiler
42      clk <= signals(65);
43      write_en <= signals(64);
44      write_inst <= signals(63 downto 32);
45      write_addr <= signals(31 downto 0);
```

An assembly program can then be written in the process of the testbench, using the producers defined in the compiler file *[figure 5.1:9]*. By constructing the testbench for the microprocessor core in this way, it is very apparent which part needs to be changed to write a different assembly program without altering the testbench itself.

Figure 5.1:9 - Assembly Program (Core_testbench.vhd)

```
47      process begin
48          -- 0
49          lui(31, 1, signals); -- x31 = 4096
50          -- 4
51          nop(signals);
52          -- 8
53          jal(1, 8, signals); -- PC = 24
54          -- 12
55          nop(signals);
56          -- 16
57          sw(30, 31, 0, signals); -- data_addr[16 + 0] = 4096 (data_addr[rs1 + imm] = rs2)
58          -- 20
59          lw(29, 30, 0, signals); -- x29 = data_addr[16 + 0] (data_addr[rs1 + imm] = rs2)
60          -- 24
61          srli(30, 31, 8, signals); -- x30 = 4096 >> 8 (x31 >> constant) = 16
62          -- 28
63          nop(signals);
64          -- 32
65          nop(signals);
```

```
66          -- 36
67          bne(30, 31, -10, signals); -- if x30 != x31, PC = 16
68          run(signals);
69      end process;
```

5.2 Test Vectors

Exhaustive testing was not used as the number of test vectors increases exponentially as the number of inputs increases. It is therefore infeasible to try all combinations given the time constraints of this project. Also, given the complexity of the circuit, structural testing that accounts for all stuck-at faults would also be far too time-consuming. Instead, the test vectors for each testbench were carefully-chosen to test all possible functions of each entity with the minimal amount of test vectors. For example, when testing the ALU *[figure 5.2:1]*, test vectors "-1 & -1" (every bit set to 1) are used for the logical AND operation. This is done to ensure that all 32 bits are operated on. However, when testing the arithmetic ADD operation, test vectors "-2147483648 + -1" (smallest number supported by 32-bit two's compliant) are used to ensure overflows are handled correctly.

Figure 5.2:1 - ALU Test Vectors

```
179         -- (operand1, operand2, expected)
180         test_add(0, 0, 0);
181         test_add(10, 10, 20);
182         test_add(-10, -10, -20);
183         test_add(-10, 10, 0);
184         test_add(2147483647, -2147483648, -1);
185         test_add(-2147483648, -1, 2147483647); -- overflow
186         test_add(2147483647, 1, -2147483648); -- overflow
187
188         test_slt(-1, 0, 1);
189         test_slt(0, -1, 0);
190         test_slt(-2147483648, 2147483647, 1);
191         test_slt(2147483647, -2147483648, 0);
192         test_slt(0, 0, 0);
193
194         test_sltu(-1, 0, 0);
195         test_sltu(0, -1, 1);
196         test_sltu(-2147483648, 2147483647, 0);
197         test_sltu(2147483647, -2147483648, 1);
198         test_sltu(0, 0, 0);
199
200         test_and(9, 13, 9);
201         test_and(-1, -1, -1); -- all 1's
202         test_and(0, 0, 0); -- all 0's
203         test_and(-1, 0, 0);
204         test_and(0, -1, 0);
205
206         test_or(9, 13, 13);
207         test_or(-1, -1, -1); -- all 1's
208         test_or(0, 0, 0); -- all 0's
209         test_or(-1, 0, -1);
210         test_or(0, -1, -1);
211
212         test_xor(9, 13, 4);
213         test_xor(-1, -1, 0);
214         test_xor(0, 0, 0);
215         test_xor(-1, 0, -1);
216         test_xor(0, -1, -1);
217
```

```
218         test_sll(5, 10, 5120);
219         test_sll(-1, -1, -2147483648);
220         test_sll(-1, 31, -2147483648);
221         test_sll(0, 0, 0);
222         test_sll(1431655765, 1, -1431655766);
223
224         test_srl(-65538, 15, 131069);
225         test_srl(-1, -1, 1);
226         test_srl(-1, 31, 1);
227         test_srl(0, 0, 0);
228         test_srl(-1431655766, 1, 1431655765);
229
230         test_sra(-1431655766, 1, -715827883);
231         test_sra(-1, -1, -1);
232         test_sra(-1, 31, -1);
233         test_sra(0, 0, 0);
234         test_sra(-715827883, 1, -357913942);
235
236         test_sub(0, 0, 0);
237         test_sub(10, 10, 0);
238         test_sub(-10, -10, 0);
239         test_sub(-10, 10, -20);
240         test_sub(-2147483648, -2147483648, 0);
241         test_sub(2147483647, 2147483647, 0);
242         test_sub(2147483647, -1, -2147483648); -- overflow
243         test_sub(-2147483648, 1, 2147483647); -- overflow
244
245         test_pass(-100, 8, 8);
```

5.3 Simulation

The implementation itself was compiled using the Intel Quartus Prime (Altera) IDE so the integrated EDA tools were used to perform all testing. The third part ModelSim simulator/compiler will be the target for the testbenches instead of Intel Quartus Prime itself. This way, the testbenches can be altered without having to recompile the entity being tested every time. Another benefit of using ModelSim is that it has a logging window *[figure 5.3:1]* and a waveform of the inputs/outputs are automatically produced *[figure 5.3:2]*. This allows the testbench to report any faults to the user in real time so they can then manually inspect the response of the entity. This can relieve useful insights into the cause of the fault so it can quickly be rectified.

Figure 5.3:1 - Example ModelSim log

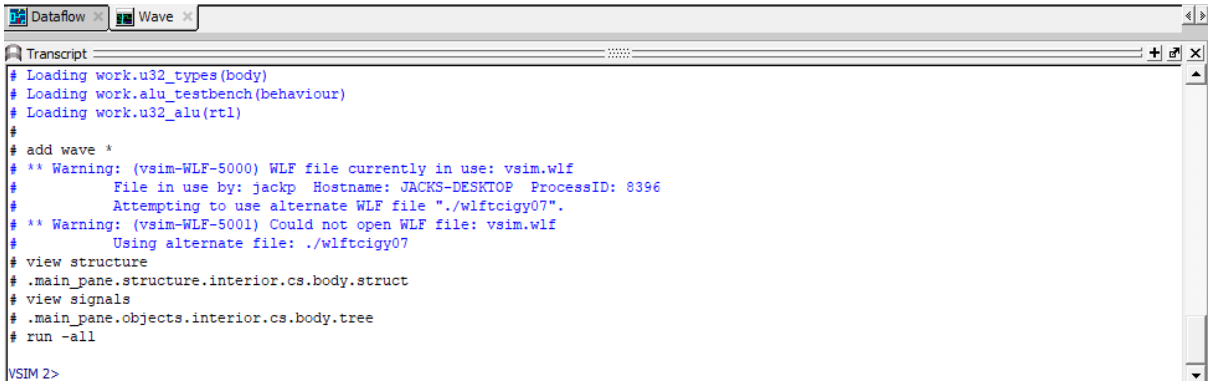
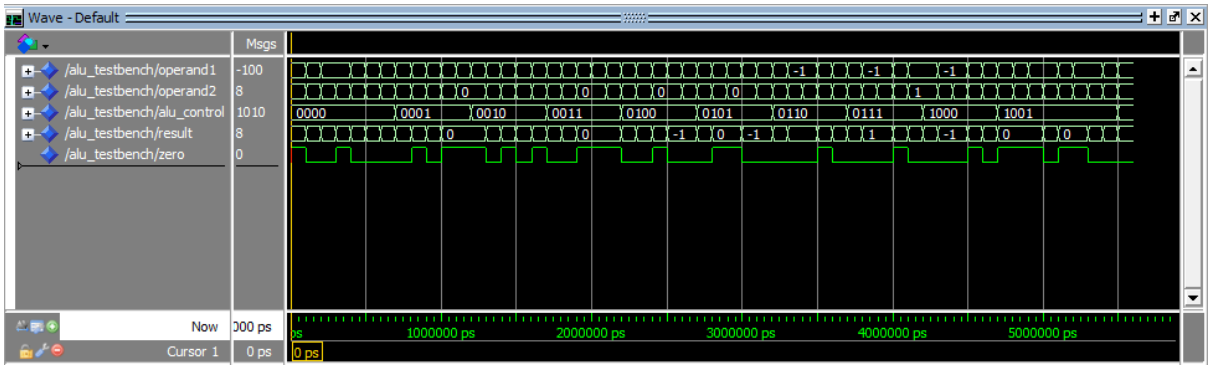


Figure 5.3:2 - Example ModelSim Waveform



There are two levels of simulation supported by ModelSim in Intel Quartus Prime, RTL and gate level. All testbenches will be run under RTL simulation and the microprocessor core testbench will be run under both RTL and gate level simulation.

6 Results

In this section, the results from the final version of the implementation detailed in section 4 will be discussed. Every little fix that was made to get to the final version will not be discussed as these were minor changes. Also, this section will focus on the pipeline stages and core as a whole, not the standalone entities, as it would be too much to cover.

6.1 Instruction Fetch

The testbench starts by filling the instruction memory (one address per clock cycle). The Inst Fetch stage doesn't depend on the instruction, so each memory location was simply filled with a 32-bit integer, where:

$$instruction = (last\ address - 4) - address$$

In this case, the instruction memory is set to 256B so:

$$instruction = 252 - address$$

In the clock cycle where the last address is filled, instructions begin to be read from the instruction memory and the pipeline registers are enabled (clk_en=1). Two cycles after that, a jump/branch is simulated on the inputs. This gives the correct operation of:

1. Finish filling instruction memory; read instruction at address 0; increment program counter (PC=PC+4)
2. Propagate values from previous cycle out of pipeline registers (pc_out=0, inst=252, etc); read instruction at address 4; increment program counter
3. Propagate values from previous cycle out of pipeline registers (pc_out=4, inst=248, etc); read instruction at address 8; simulate jump/branch (pc_src=1 and new_pc=36)
4. Propagate values from previous cycle out of pipeline registers (pc_out=8, inst=244, etc); read instruction at address 36; increment program counter
5. Propagate values from previous cycle out of pipeline registers (pc_out=36, inst=216, etc); read instruction at address 40; finish

When running the testbench at RTL, no errors were output in the simulation log *[figure 6.1:2]*. Observing the waveform *[figure 6.1:1]*, shows that the final implementation of the Inst Fetch pipeline behaves correctly.

Figure 6.1:1 - Inst Fetch RTL Waveform

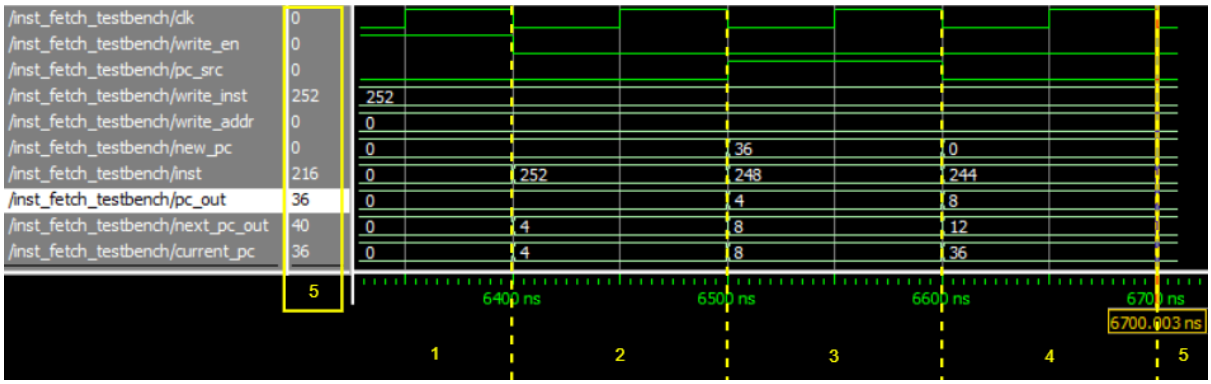


Figure 6.1:2 - Inst Fetch RTL Log

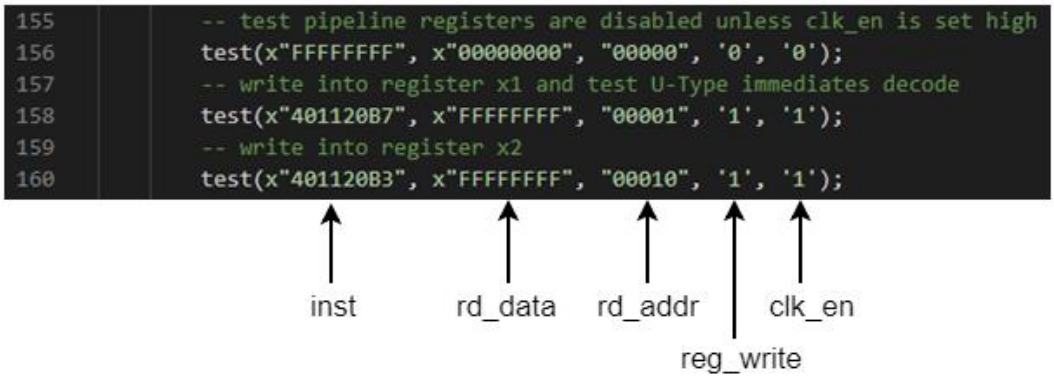
```
Transcript
# vsim -t 1ps -L altera -L lpm -L sgate -L altera_mf -L altera_lnsim -L cycloneive -L rtl_work -L work -voptargs=""+acc"" inst_fetch_testbench
# Start time: 12:47:49 on Apr 27,2019
# Loading std.standard
# Loading std.textio(body)
# Loading ieee.std_logic_1164(body)
# Loading ieee.std_logic_arith(body)
# Loading ieee.std_logic_unsigned(body)
# Loading ieee.numeric_std(body)
# Loading work.u32_types(body)
# Loading work.inst_fetch_testbench(behaviour)
# Loading work.u32_inst_fetch(rtl)
#
# add wave *
# ** Warning: (vsim-WLF-5000) WLF file currently in use: vsim.wlf
#           File in use by: jackp  Hostname: JACKS-DESKTOP  ProcessID: 8396
#           Attempting to use alternate WLF file "./wlfteyr94z".
# ** Warning: (vsim-WLF-5001) Could not open WLF file: vsim.wlf
#           Using alternate file: ./wlfteyr94z
# view structure
# .main_pane.structure.interior.cs.body.struct
# view signals
# .main_pane.objects.interior.cs.body.tree
# run -all

VSIM 2>
```

6.2 Decode

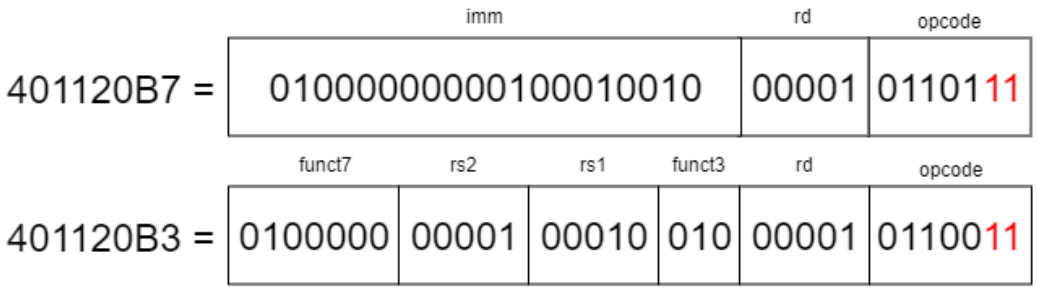
As the Decode stage consists only of instantiated entities and pipeline registers, the instantiated entities will have already passed testing before being instantiated into the decode stage. Therefore, the testbench for the decode pipeline stage only needs to test that the instantiated entities, and pipeline registers have been wired up correctly. To do this, three test vectors were carefully chosen *[figure 6.2:1]*.

Figure 6.2:1 - Decode Test Vectors



The first test vector is used to check if **clk_en** stalls the pipeline registers so the instruction used does not matter. The other two instruction are as follows:

Figure 6.2:2 - Decode Test Instructions



This gives the correct operation of:

1. Stall pipeline registers
2. Decode immediate (zero pad x“40112”); decode control signals from (opcode = “01101”); write data (x”FFFFFFFF”) into register x1
3. Propagate values from previous cycle out of pipeline registers (imm=x”40112000”, control=”100001000”, etc); decode control signals from (opcode = “01100”); write data (x”FFFFFFFF”) into register x2; read data from registers x1 & x2
4. Propagate values from previous cycle out of pipeline registers (rs1d=x”FFFFFFFF”, rs1d=x”FFFFFFFF”, control=”100101100”, etc); finish

The control output is checked against a look-up table which details the control signals that should be produced given a certain opcode. Therefore, any issues with the pipeline controller would be logged in the simulators log *[figure 6.2:4]*. Observing the waveform *[figure 6.2:3]* shows that the final implementation of the Decode stage behaves correctly.

Figure 6.2:3 - Decode RTL Waveform

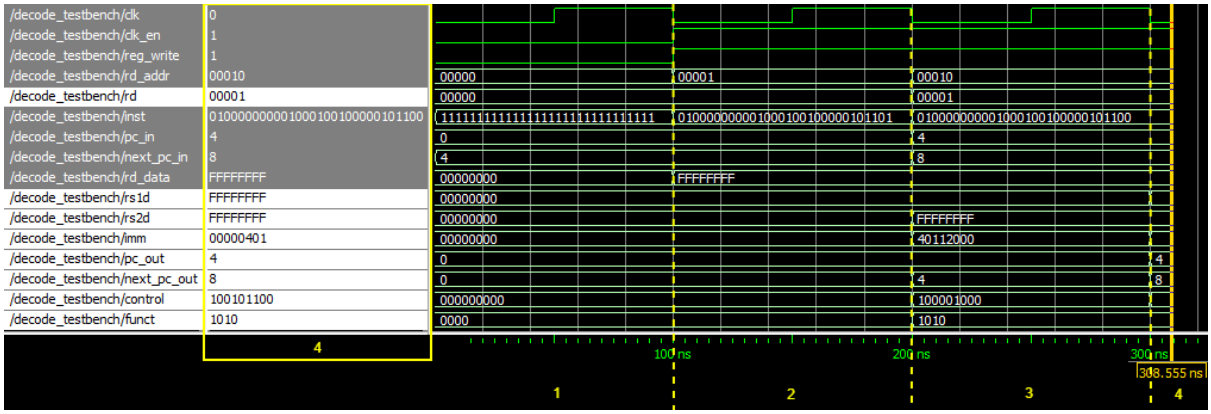


Figure 6.2:4 - Decode RTL Log

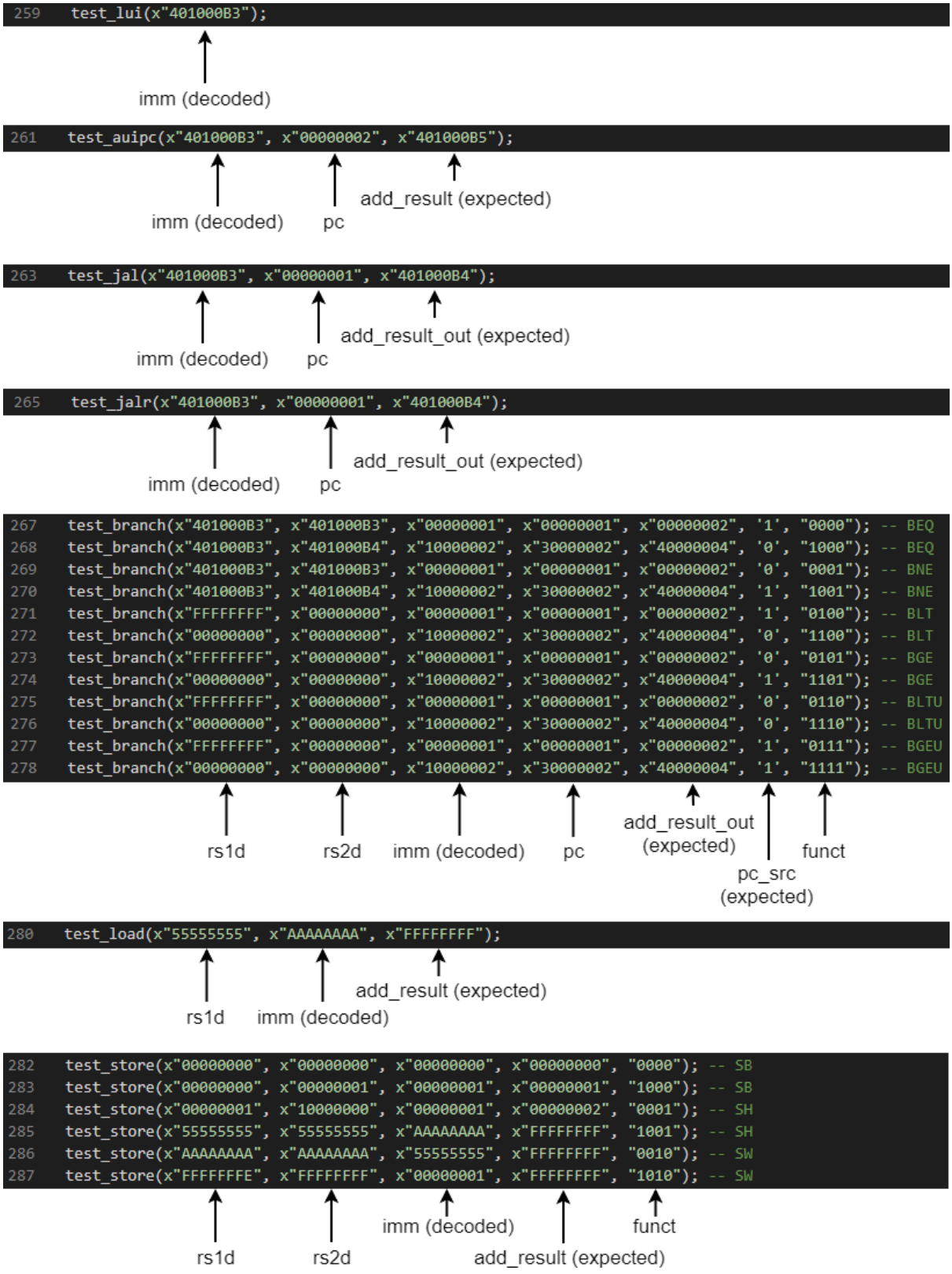
```
Transcript
# vsim -t 1ps -L altera -L lpm -L sgate -L altera_mf -L altera_lnsim -L cycloneive -L rtl_work -L work -voptargs=""+acc"" decode_testbench
# Start time: 12:53:14 on Apr 27,2019
# Loading std.standard
# Loading std.textio(body)
# Loading ieee.std_logic_1164(body)
# Loading ieee.std_logic_arith(body)
# Loading ieee.std_logic_unsigned(body)
# Loading ieee.numeric_std(body)
# Loading work.u32_types(body)
# Loading work.decode_testbench(behaviour)
# Loading work.u32_decode(rtl)
# Loading work.u32_controller(rtl)
# Loading work.u32_gp_registers(rtl)
# Loading work.u32_immediate_decoder(rtl)
#
# add wave *
# ** Warning: (vsim-WLF-5000) WLF file currently in use: vsim.wlf
#           File in use by: jackp  Hostname: JACKS-DESKTOP  ProcessID: 8396
#           Attempting to use alternate WLF file "./wlftgxt8nj".
# ** Warning: (vsim-WLF-5001) Could not open WLF file: vsim.wlf
#           Using alternate file: ./wlftgxt8nj
# view structure
# .main_pane.structure.interior.cs.body.struct
# view signals
# .main_pane.objects.interior.cs.body.tree
# run -all
# ** Note: Tested inst: rs1d = 0x00000000; rs2d = 0x00000000; funct = 0000; imm = 0x00000000; rd = 00000;
#    Time: 100001 ps  Iteration: 0  Instance: /decode_testbench
# ** Note: Tested inst: rs1d = 0x00000000; rs2d = 0xFFFFFFFF; funct = 1010; imm = 0x40112000; rd = 00001;
#    Time: 200002 ps  Iteration: 0  Instance: /decode_testbench
# ** Note: Tested inst: rs1d = 0xFFFFFFFF; rs2d = 0xFFFFFFFF; funct = 1010; imm = 0x00000401; rd = 00001;
#    Time: 300003 ps  Iteration: 0  Instance: /decode_testbench

VSIM 2>
```

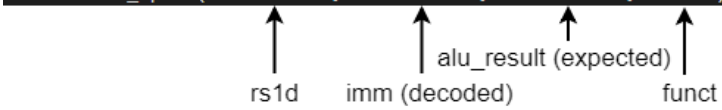
6.3 Execute

The Execute stage is the most complex to test as it has the highest number of inputs and multiplexers. It, therefore, has considerably more test vectors than any other stage *[figure 6.3:1]*. The testbench sets the inputs as if instructions had already passed through the decode stage.

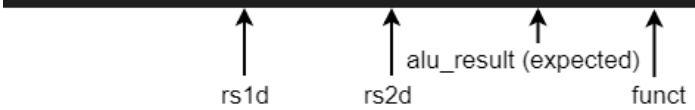
Figure 6.3:1 - Execute Test Vectors



```
289   test_opimm(x"55555555", x"AAAAAAAA", x"FFFFFFFF", "0000"); -- ADDI
290   test_opimm(x"55555555", x"AAAAAAAA", x"FFFFFFFF", "1000"); -- ADDI
291   test_opimm(x"FFFFFFFF", x"00000000", x"00000001", "0010"); -- SLTI
292   test_opimm(x"FFFFFFFF", x"00000000", x"00000001", "1010"); -- SLTI
293   test_opimm(x"FFFFFFFF", x"00000000", x"00000000", "0011"); -- SLTUI
294   test_opimm(x"FFFFFFFF", x"00000000", x"00000000", "1011"); -- SLTUI
295   test_opimm(x"FFFFFFFF", x"FFFFFFFF", x"00000000", "0100"); -- XORI
296   test_opimm(x"FFFFFFFF", x"FFFFFFFF", x"00000000", "1100"); -- XORI
297   test_opimm(x"FFFFFFFF", x"00000000", x"FFFFFFFF", "0110"); -- ORI
298   test_opimm(x"FFFFFFFF", x"00000000", x"FFFFFFFF", "1110"); -- ORI
299   test_opimm(x"FFFFFFFF", x"00000000", x"00000000", "0111"); -- ANDI
300   test_opimm(x"FFFFFFFF", x"00000000", x"00000000", "1111"); -- ANDI
301   test_opimm(x"55555555", x"00000001", x"AAAAAAAA", "0001"); -- SLLI
302   test_opimm(x"AAAAAAAA", x"00000001", x"55555555", "0101"); -- SRLI
303   test_opimm(x"AAAAAAAA", x"00000001", x"D5555555", "1101"); -- SRAI
```




```
305   test_op(x"55555555", x"AAAAAAAA", x"FFFFFFFF", "0000"); -- ADD
306   test_op(x"FFFFFFFF", x"55555555", x"AAAAAAAA", "1000"); -- SUB
307   test_op(x"55555555", x"00000001", x"AAAAAAAA", "0001"); -- SLL
308   test_op(x"FFFFFFFF", x"00000000", x"00000001", "0010"); -- SLT
309   test_op(x"FFFFFFFF", x"00000000", x"00000000", "0011"); -- SLTU
310   test_op(x"FFFFFFFF", x"FFFFFFFF", x"00000000", "0100"); -- XOR
311   test_op(x"AAAAAAAA", x"00000001", x"55555555", "0101"); -- SRL
312   test_op(x"AAAAAAAA", x"00000001", x"D5555555", "1101"); -- SRA
313   test_op(x"FFFFFFFF", x"00000000", x"FFFFFFFF", "0110"); -- OR
314   test_op(x"FFFFFFFF", x"00000000", x"00000000", "0111"); -- AND
```



This many test vectors means the waveform is too complex to be of much use, unless an issue (result doesn't equal expected) is logged and the exact time the issue occurs is known. Observing the RTL simulation log *[figure 6.2:3]* shows that no errors were logged so the final implementation of the Execute stage behaves correctly.

Figure 6.3:2 - Execute RTL Log

```
Transcript
# vsim -t 1ps -L altera -L lpm -L sgate -L altera_mf -L altera_lnsim -L cycloneive -L rtl_work -L work -voptargs=""+acc"" execute_testbench
# Start time: 12:44:34 on Apr 27,2019
# Loading std.standard
# Loading std.textio(body)
# Loading ieee.std_logic_1164(body)
# Loading ieee.std_logic_arith(body)
# Loading ieee.std_logic_unsigned(body)
# Loading ieee.std_logic_signed(body)
# Loading ieee.numeric_std(body)
# Loading work.u32_types(body)
# Loading work.execute_testbench(behaviour)
# Loading work.u32_execute(rtl)
# Loading work.u32_alu(rtl)
# Loading work.u32_alu_controller(rtl)
# Loading work.u32_branch_controller(rtl)
#
# add wave *
# ** Warning: (vsim-WLF-5000) WLF file currently in use: vsim.wlf
#           File in use by: jackp  Hostname: JACKS-DESKTOP  ProcessID: 8396
#           Attempting to use alternate WLF file "./wlftw2nhw5".
# ** Warning: (vsim-WLF-5001) Could not open WLF file: vsim.wlf
#           Using alternate file: ./wlftw2nhw5
# view structure
# .main_pane.structure.interior.cs.body.struct
# view signals
# .main_pane.objects.interior.cs.body.tree
# run -all

VSIM 2>
```

6.4 Memory Access

The Mem Access stage is another that doesn't require many test vectors *[figure 6.4:1]*, as most instructions simply propagate through this stage, and all instructions with the BRANCH opcode do not use this stage at all. In fact, only instructions with STORE or LOAD opcodes are operated on in this stage. Instructions with the LUI, JAL or JALR opcode have the same control signals at this stage in the pipeline, so are propagated through in the same exact way. Similarly, instructions that use the OP or OP-IMM opcode have the same control signals at this stage. Also, the data memory will have already passed testing before being instantiated into the Mem Access stage.

Figure 6.4:1 - Mem Access Test Vectors

```
138         test_lui_jal_jalr; -- check the addr_const is propgated output of funct_rdd
139
140         test_auipc;  -- check the add_result is propgated output of funct_rdd
141
142         test_op_opimm;  -- check the alu_result is propgated output of funct_rdd
143
144         test_load(x"00000004", x"00000000"); -- check no data stored in byte address 4
145
146         test_store(x"00000004", x"AAAAAAAA"); -- store data in byte address 4
147
148         test_load(x"00000004", x"AAAAAAAA"); -- check data stored in byte address 4
```

This gives the correct operation of:

1. Passthrough addr_const (x"55555555")
2. Propagate values from previous cycle out of pipeline registers (funct_rdd =x"55555555", control="10", etc); Passthrough add_result (x"000000AA")
3. Propagate values from previous cycle out of pipeline registers (funct_rdd =x" 000000AA", control="10", etc); Passthrough alu_result (x"FFFFFFFF")
4. Propagate values from previous cycle out of pipeline registers (funct_rdd =x" FFFFFFFF", control="10", etc); read data from memory address 4; Passthrough funct ("1110" zero extended = x"0000000E")
5. Propagate values from previous cycle out of pipeline registers (oper=x"00000000", funct_rdd =x"0000000E", control="11", etc); write data (x"AAAAAAAA") into memory address 4
6. Read data from memory address 4; Passthrough funct ("1110" zero extended = x"0000000E")
7. Propagate values from previous cycle out of pipeline registers (oper=x" AAAAAAAA", funct_rdd =x"0000000E", control="11", etc)

Observing the waveform *[figure 6.4:3]*, shows that the final implementation of the Mem Access stage behaves correctly.

Figure 6.4:2 - Mem Access Waveform

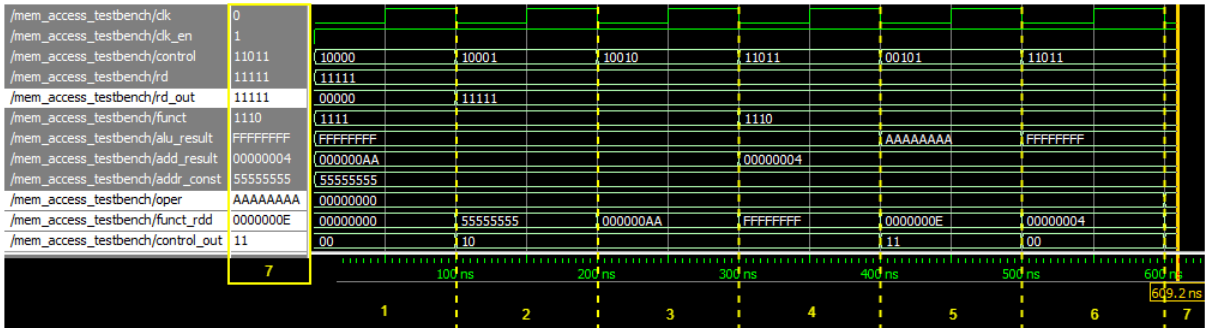


Figure 6.4:3 - Mem Access RTL Log

```
Transcript
# vsim -t 1ps -L altera -L lpm -L sgate -L altera_mf -L altera_lnsim -L cycloneive -L rtl_work -L work -voptargs=""+acc"" mem_access_testbench
# Start time: 14:16:06 on Apr 27,2019
# Loading std.standard
# Loading std.textio(body)
# Loading ieee.std_logic_1164(body)
# Loading ieee.std_logic_arith(body)
# Loading ieee.std_logic_unsigned(body)
# Loading ieee.numeric_std(body)
# Loading work.u32_types(body)
# Loading work.mem_access_testbench(behaviour)
# Loading work.u32_mem_access(rtl)
# Loading work.u32_data_memory(behavioral)
#
# add wave *
# ** Warning: (vsim-WLF-5000) WLF file currently in use: vsim.wlf
#           File in use by: jackp  Hostname: JACKS-DESKTOP  ProcessID: 8396
#           Attempting to use alternate WLF file "./wlftckqg2m".
# ** Warning: (vsim-WLF-5001) Could not open WLF file: vsim.wlf
#           Using alternate file: ./wlftckqg2m
# view structure
# .main_pane.structure.interior.cs.body.struct
# view signals
# .main_pane.objects.interior.cs.body.tree
# run -all

VSIM 2>
```

6.5 Write Back

The writeback stage was not tested on its own as it consists only of an instantiated entity that has already passed testing and a single multiplexer. Instead, the wiring of the schematic shown in section 4.6 was checked against the design shown in section 3.3.1. Doing this showed that the instantiated entity, multiplexer, inputs, and outputs were all wired up correctly.

6.6 Core

After all pipeline stages were proven to be operating correctly under RTL simulation, they were all combined to form the final implementation of the microprocessor core. The testbench detailed in section 5.1.2 was used to run a number of assembly programs on the core. The first few programs were very simple and were found to operate as expected. However, these tests weren't very useful as they only tested one or two types of instruction at a time. To properly test the core a program was written *[figure 6.6:1]* which tested each type of instruction (integer arithmetic, load/store, control transfer) at least once. All these instructions operated on the same data, that way if any instruction executed incorrectly, the final result would change. This would make it very easy to determine if the program had executed correctly or not.

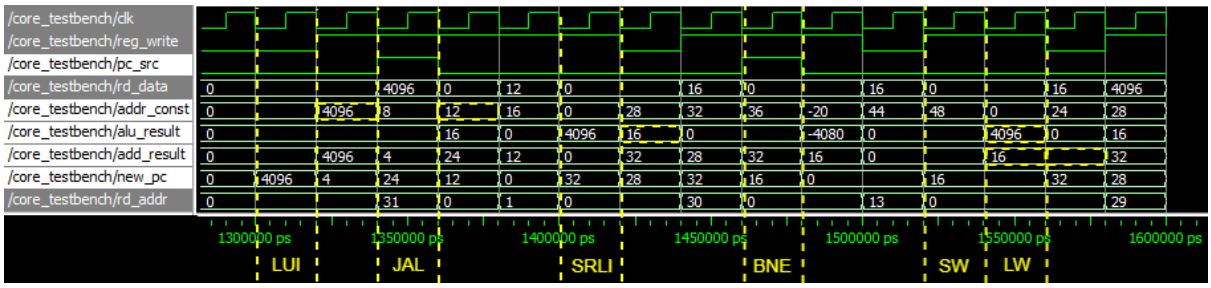
Figure 6.6:1 - Core Test Program

```
48      -- 0
49      lui(31, 1, signals); -- x31 = 4096 (imm<<12)
50      -- 4
51      nop(signals);
52      -- 8
53      jal(1, 8, signals); -- PC = 24 (PC + (2 * imm))
54      -- 12
55      nop(signals);
56      -- 16
57      sw(30, 31, 0, signals); -- data_addr[16 + 0] = 4096 (data_addr[rs1 + imm] = rs2)
58      -- 20
59      lw(29, 30, 0, signals); -- x29 = data_addr[16 + 0] (data_addr[rs1 + imm] = rs2)
60      -- 24
61      srli(30, 31, 8, signals); -- x30 = 4096 >> 8 (x31 >> imm) = 16
62      -- 28
63      nop(signals);
64      -- 32
65      nop(signals);
66      -- 36
67      bne(30, 31, -10, signals); -- if x30 != x31, PC = 16 (PC + (2 * imm))
68      run(signals);
```

The ImperialViolet RISC-V assembly language [10] is a good reference for understanding how this program works. The core can execute all of the instructions detailed in that language, with the expression of the pseudo-instructions. The only pseudo instruction defined in the testbench is the NOP instruction. The NOP instruction is used throughout the program as the core does not contain any stall/flush logic. It is, therefore, down to the programmer to insert pipeline bubbles in the form of NOP instructions, to alleviate any data/control hazards.

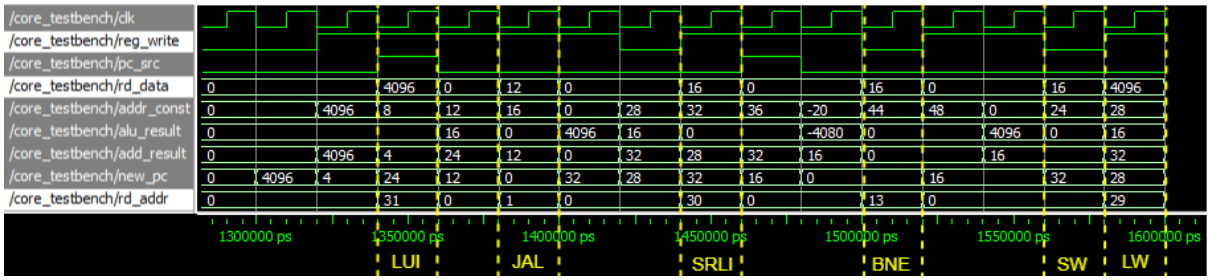
To simplify the observation of the waveform, the writing of the instructions into instruction memory was ignored as it has already been proven to work in section 6.1. To further simplify the observation, the outputs from each stage were taken separately. When observing the Execute stage outputs on the waveform *[figure 6.6:2]*, the results for the alu_result, add_result and addr_const outputs should be taken in the next clock cycle, as they need to propagate out of the pipeline registers.

Figure 6.6:2 – Core Test Program Waveform (Execute)



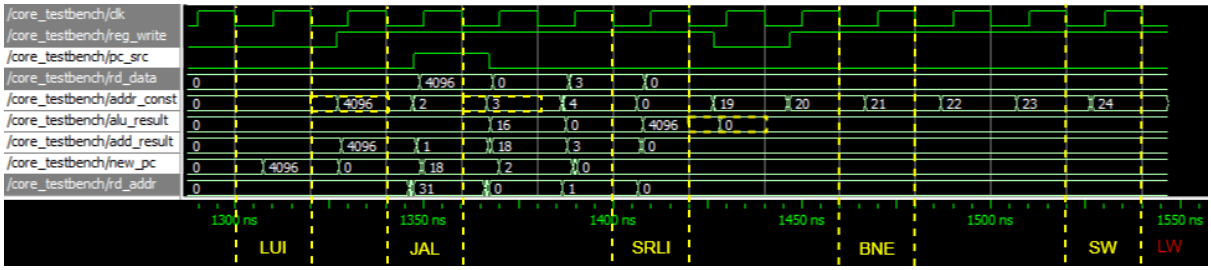
When observing the Write Back stage outputs on the waveform *[figure 6.6:3]*, there are many times were reg_write is set high when it looks as if it shouldn't be e.g. on a NOP. However, rd_addr is always 0 in those cases, and therefore does not write anything into the general-purpose registers. Knowing that and seeing that all the other outputs are as expected, the final implementation of the microprocessor core has passed all RTL simulation tests.

Figure 6.6:3 - Core Test Program RTL Waveform (Write Back)



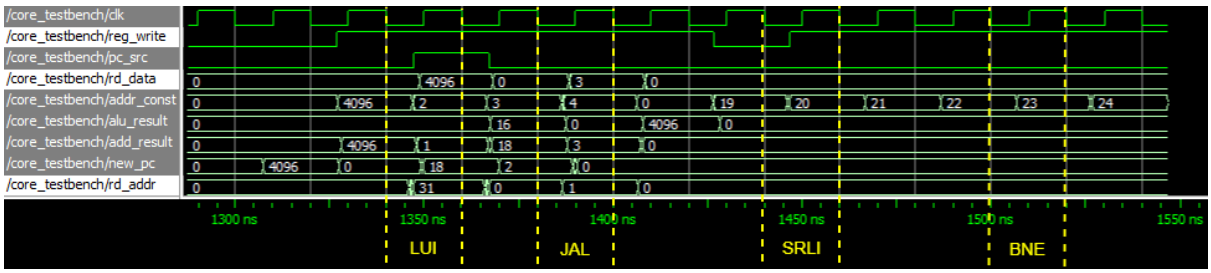
The final implementation of the microprocessor core was then tested running the same program but in gate level simulation. When observing the Execute stage outputs on the waveform *[figure 6.6:4]*, the LUI instruction produces the correct output on addr_const (4096). However, all other instructions produce unexpected results.

Figure 6.6:4 - Core Test Program Gate Level Waveform (Execute)



When observing the Write Back stage outputs on the waveform *[figure 6.6:5]*, again the LUI instruction is writing 4096 into register x31 as it should, but all other instructions produce unexpected results. This means that the final implementation cannot be uploaded onto a FPGA development board without further optimisation.

Figure 6.6:5 - Core Test Program Gate Level Waveform (Write Back)



7 Discussion

7.1 Appraisal

The initial objectives and deliverables were perhaps a little too ambitious, given the time constraints and somewhat limited resources. To meet the final objective presented in section 1.3, the final implementation needed to be uploaded and tested on a FPGA development board. As seen in section 6.6, the final implementation did not pass gate level simulation tests, so could not be uploaded onto a FPGA development board. With the testing results obtained, it's difficult to say which parts of the final implementation are causing it to fail. This can be put down to the testing methodology used. Had every single testbench been run under gate-level simulation, the issue would have been observed earlier on in the implementation and most likely resolved before testing the core as a whole.

The RTL simulation results prove that the design detailed in section 3 is capable of executing RISC-V Integer Computation, Load/Store and Control Transfer instructions. This means that the first objective and deliverable has been fully met. The second and third deliverables, have also been met as all the VHDL code is synthesisable and all simulated test results have been presented. The second and third objectives have mostly been met as the core has been implemented in VHDL and tested in simulations. However, these objectives have not been fully met as the second objective states that it is intended for use in FPGAs, and the implementation will not work on a FPGA in its current form.

Appraising the objectives individually can make it seem as if much has not been achieved. However, overall this project has made significant strides towards completing what it initially set out to achieve. The aim was to design a microprocessor core that demonstrates common microarchitecture techniques, utilises the RISC-V ISA and is simple enough to be used as an educational tool. As evidenced throughout this dissertation, every design decision was made with the aim of being used as an education tool. For example, the classic 5-stage pipeline was used as it has plenty of existing material supporting it, the RV32I instruction set was used as it's the simplest defined in the RISC-V ISA that has been frozen. Another example is the VHDL code being written with readability as the highest priority. Even without the implementation working on a FPGA, the design/implementation in combination with this dissertation is still a useful education tool and therefore meets the aim of the project.

7.2 Further Work

Despite the aim of the project being met, there is plenty of further work that this project could benefit from, given the extra time and/or resources. First, the implementation would need to be optimised to ensure that it can be used on FPGAs. Besides optimising the implementation to work on FPGA, from an education perspective, it would be good to break the implementation up so it's easier to digest. This would allow students to start with the simplest single cycle architecture and build up to something like the implementation presented in this dissertation. This would give students a better understanding of how implementation techniques, such as pipelining affect the characteristics of the implementation e.g. performance, size, power efficiency, etc.

Extending the design of this implementation, such that it is compatible with open source RISC-V assembly, C and C++ compilers would benefit this program more than anything else. To do this, the

microarchitecture would have to be able to execute the full RV32I instruction set and would need to abstract pipeline data/control hazards away from the programmer. The simplest way to do this would be to include stall/flush logic, which would stall the pipeline for two clock cycles and flush the appropriate pipeline stages whenever a data/control hazard occurs. However, a better approach would be to use simpler stall/flush logic that only stalls the pipeline on control hazards and a data forwarding unit to handle data hazards. This would improve the performance of the design and would demonstrate yet another common microarchitecture implementation technique, further improving its effectiveness as an education tool.

The potential for expansion is almost limitless and as the RISC-V ISA is used, it makes perfect sense to open source this project. First of all, it would make it accessible to anyone interested in computer architecture and organisation. This would hopefully build up an engineering community around this RISC-V implementation, that could help to improve the design and implementation far beyond what would otherwise be possible. The likelihood of long-term support would also increase by going open source. All this mean it's more likely to be adopted by universities and therefore more useful for students as an educational tool.

7.3 Conclusion

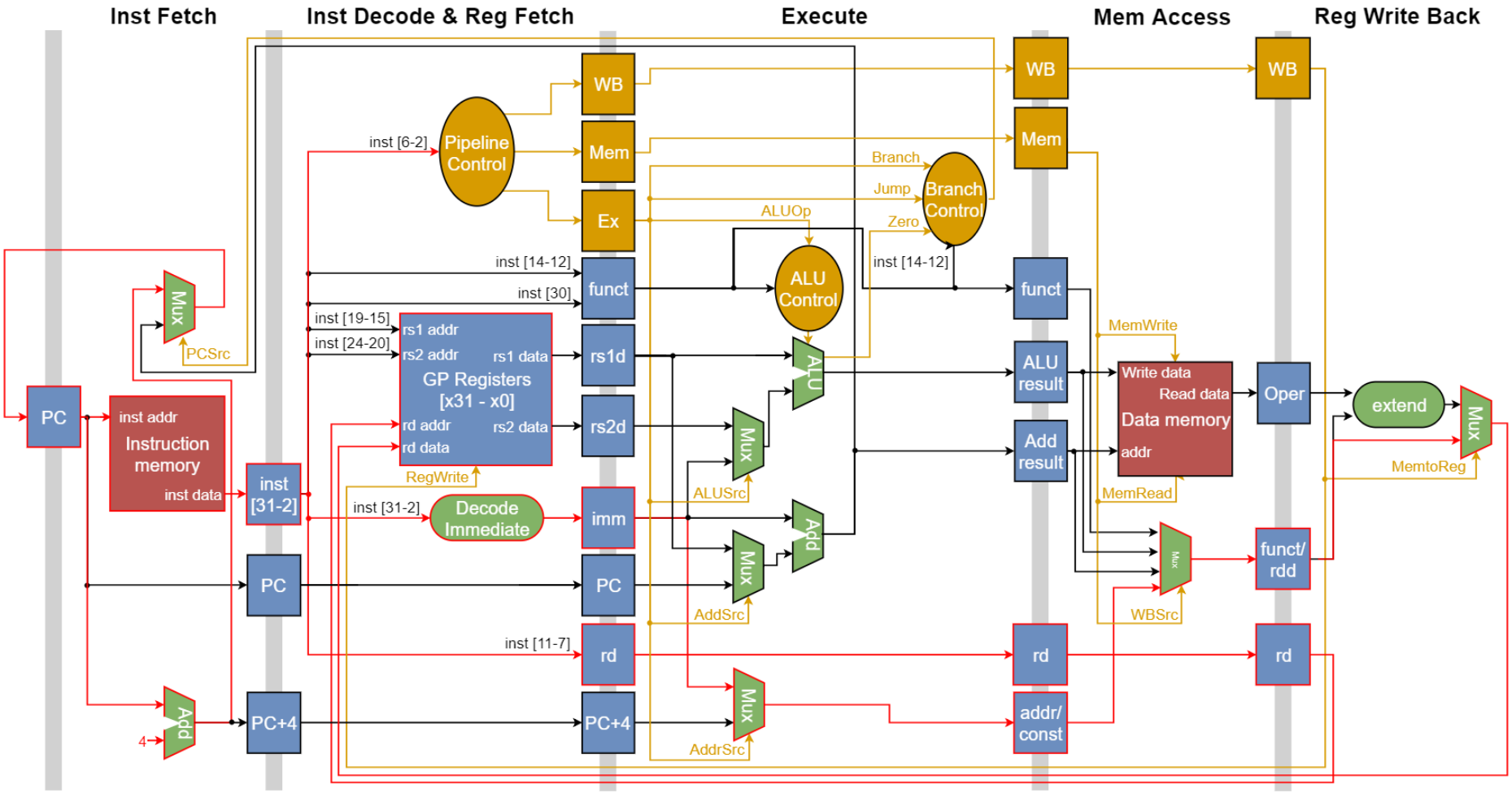
The soft RISC-V core design manages to demonstrate many microarchitecture implementation techniques while still being relatively simple in its overall design. The VHDL implementation correctly executes all RV32I Integer Computation, Load/Store and Control Transfer instructions at the RTL. In its current form, it does not execute these instructions at the gate level, meaning it will not work on a FPGA. Never the less, the combination of the design, implementation and this dissertation could still serve as a very useful educational tool. This project can, therefore, be deemed a success but given the additional time and resources, it could greatly benefit from additional work.

8 References

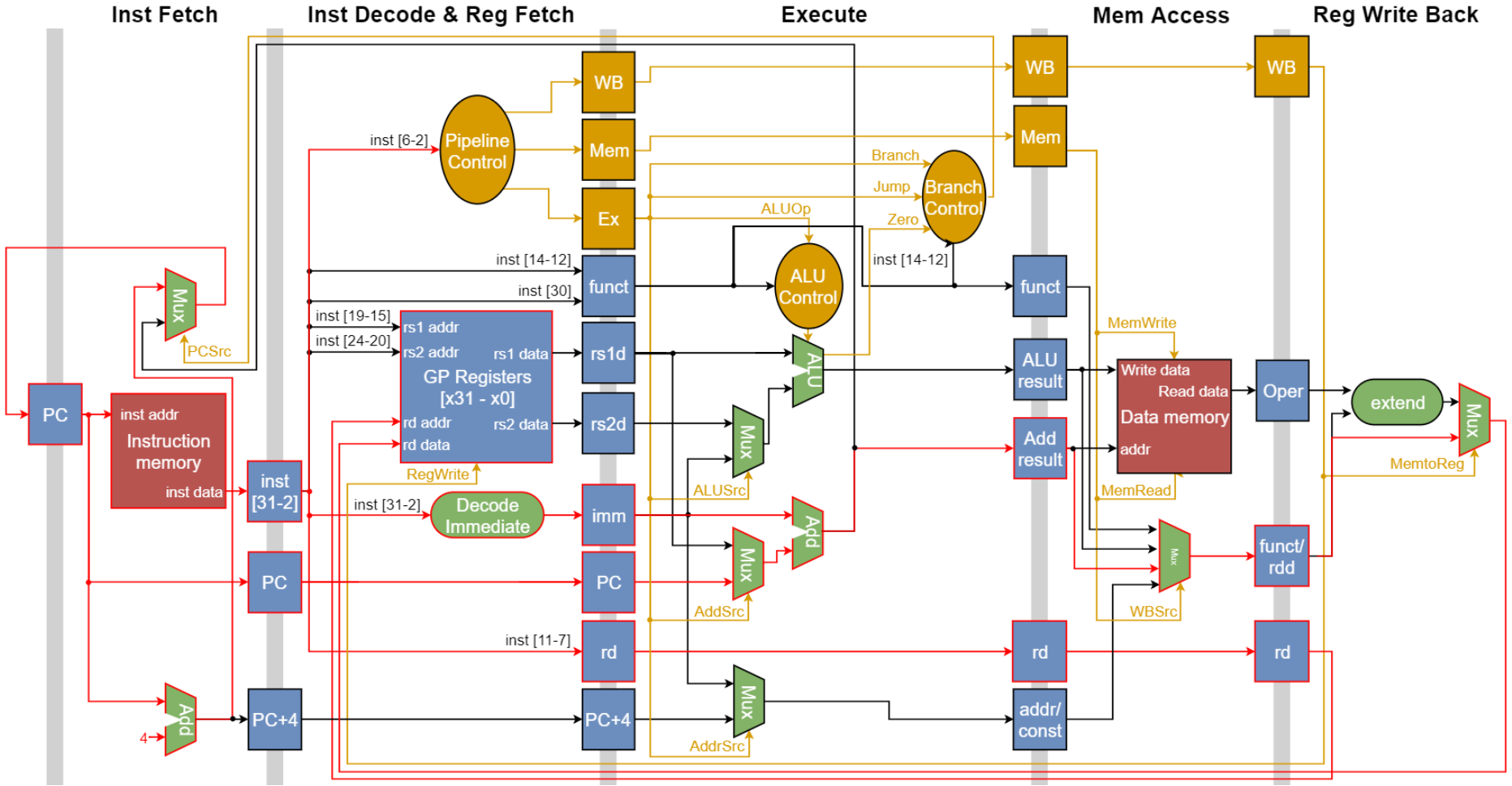
1. https://static.docs.arm.com/ddi0487/da/DDI0487D_a_armv8_arm.pdf?_ga=2.146922551.183937137.1554545416-990076710.1551448705
2. https://www-user.tu-chemnitz.de/~heha/viewchm.php/hs/x86.chm/x86.htm
3. https://passlab.github.io/CSE564/notes/lecture04_ISA_Principles.pdf
4. http://www.archive.ece.cmu.edu/~ece447/s13/lib/exe/fetch.php?media=onur-447-spring13-lecture3-isa-tradeoffs-afterlecture.pdf
5. http://db.cs.duke.edu/courses/compsci250/cps104/fall98/lectures/week14-l2/sld033.htm
6. http://web.cs.iastate.edu/~prabhu/Tutorial/PIPELINE/dataHaz.html
7. https://github.com/riscv/riscv-isa-manual/blob/master/release/riscv-spec-v2.2.pdf
8. https://www.intel.com/content/www/us/en/programmable/documentation/sbc1513987577203.html#mwh1409959584770
9. https://moodle.epfl.ch/pluginfile.php/1772382/mod_resource/content/3/vhdl_testbench_tutorial.pdf
10. https://www.imperialviolet.org/2016/12/31/riscv.html

9 Appendix A

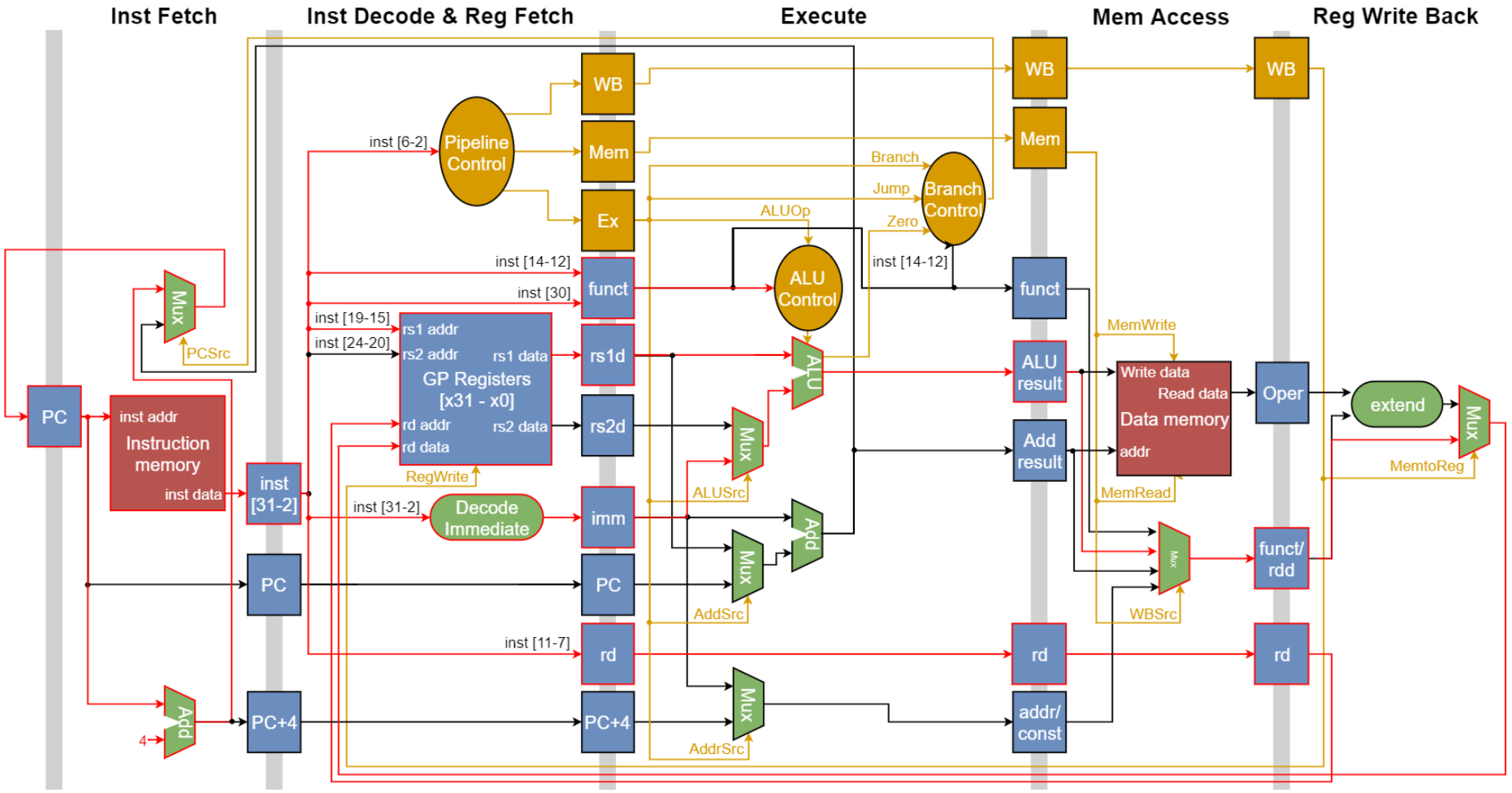
9.1 LUI Control & Data Flow



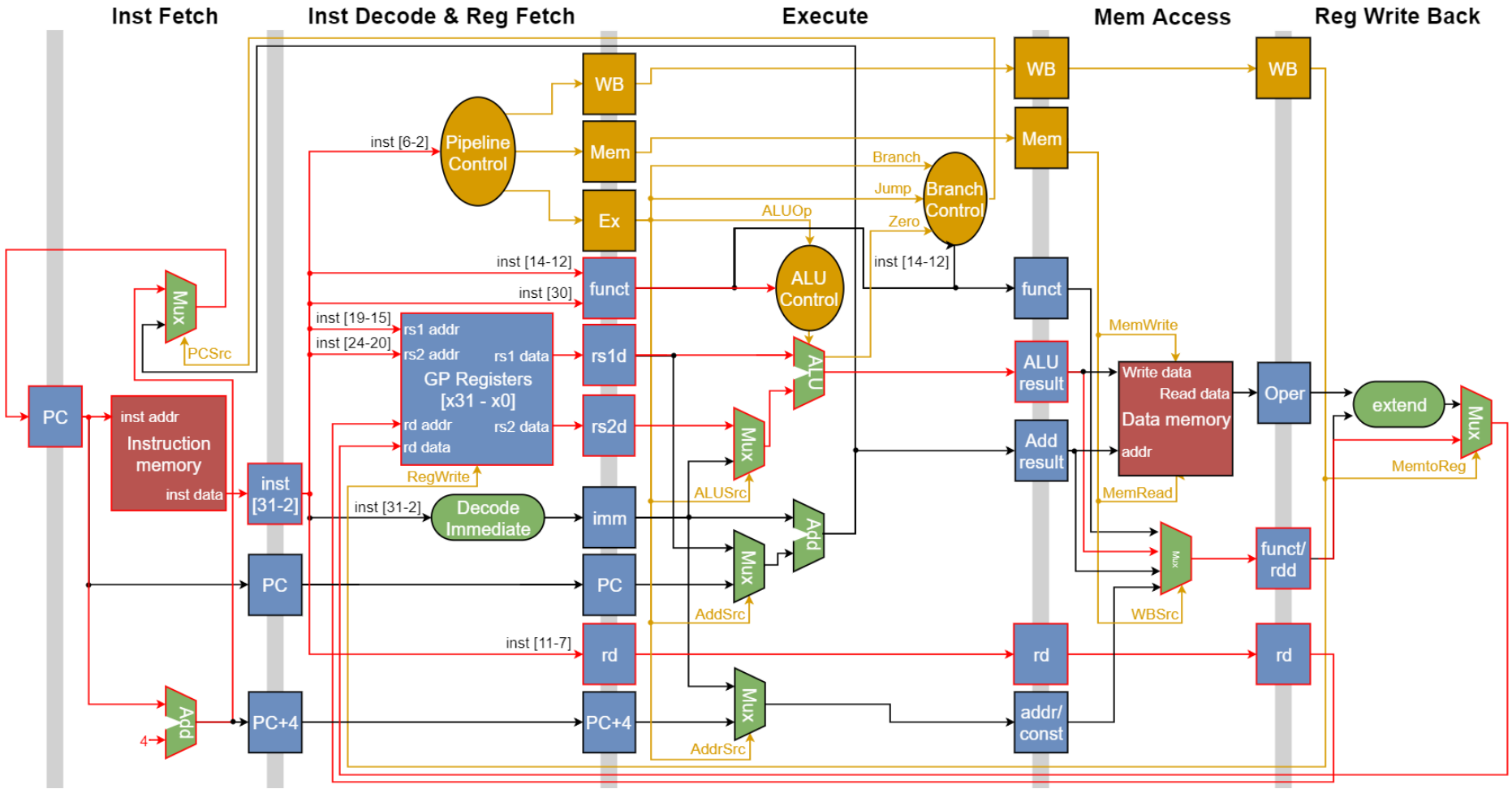
9.2 AUIPC Control & Data Flow



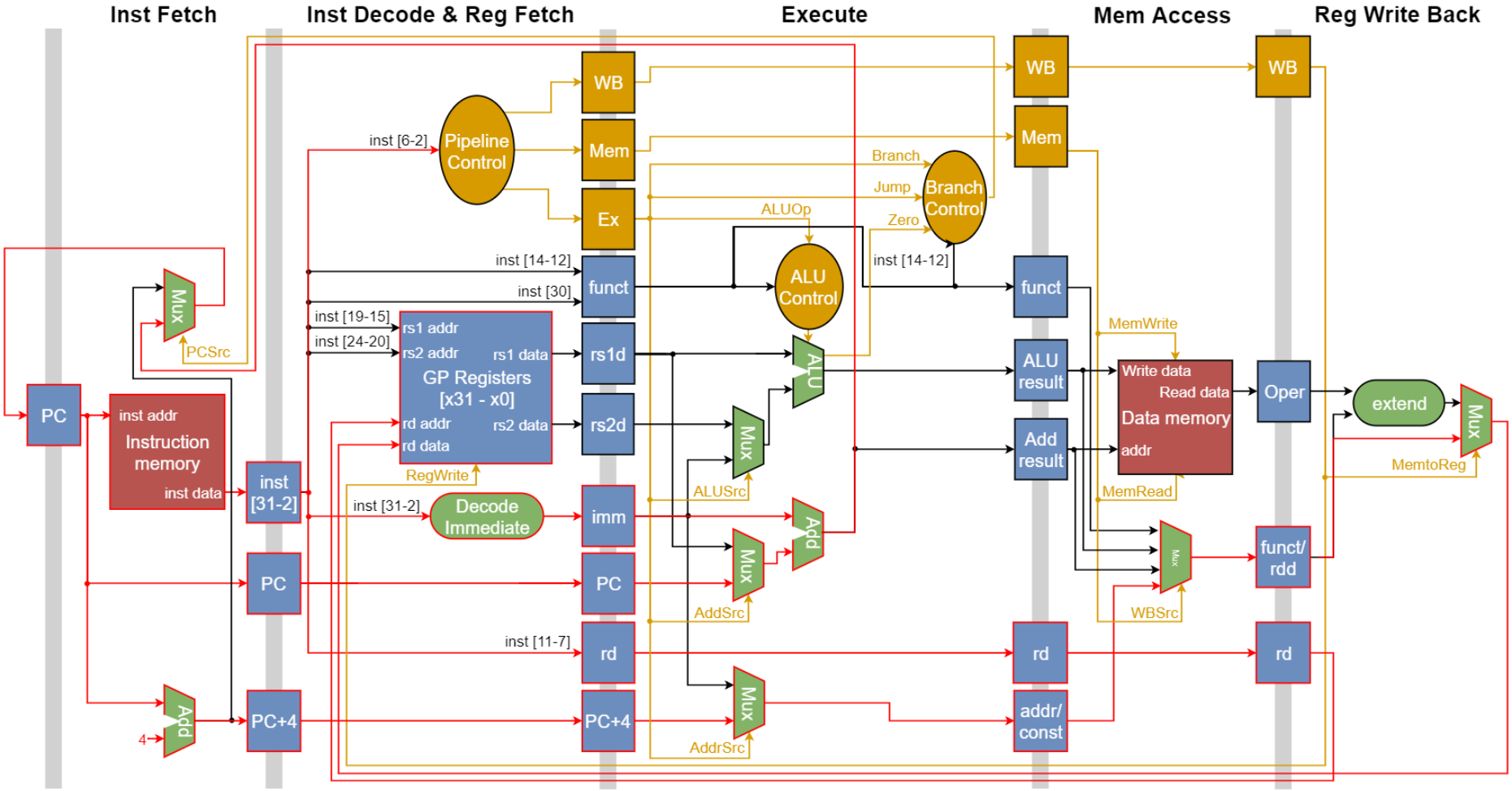
9.3 OP-IMM Control & Data Flow



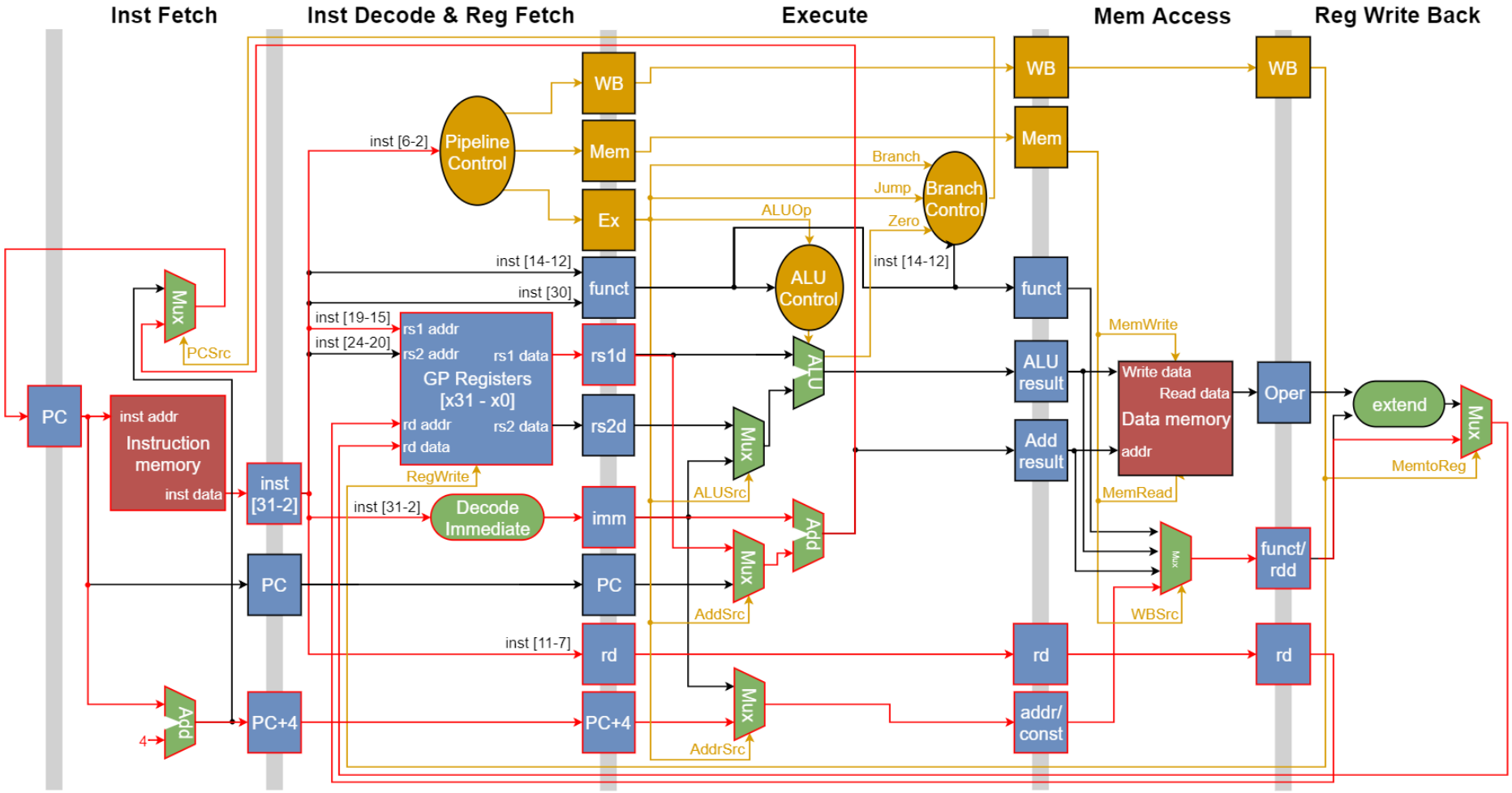
9.4 OP Control & Data Flow



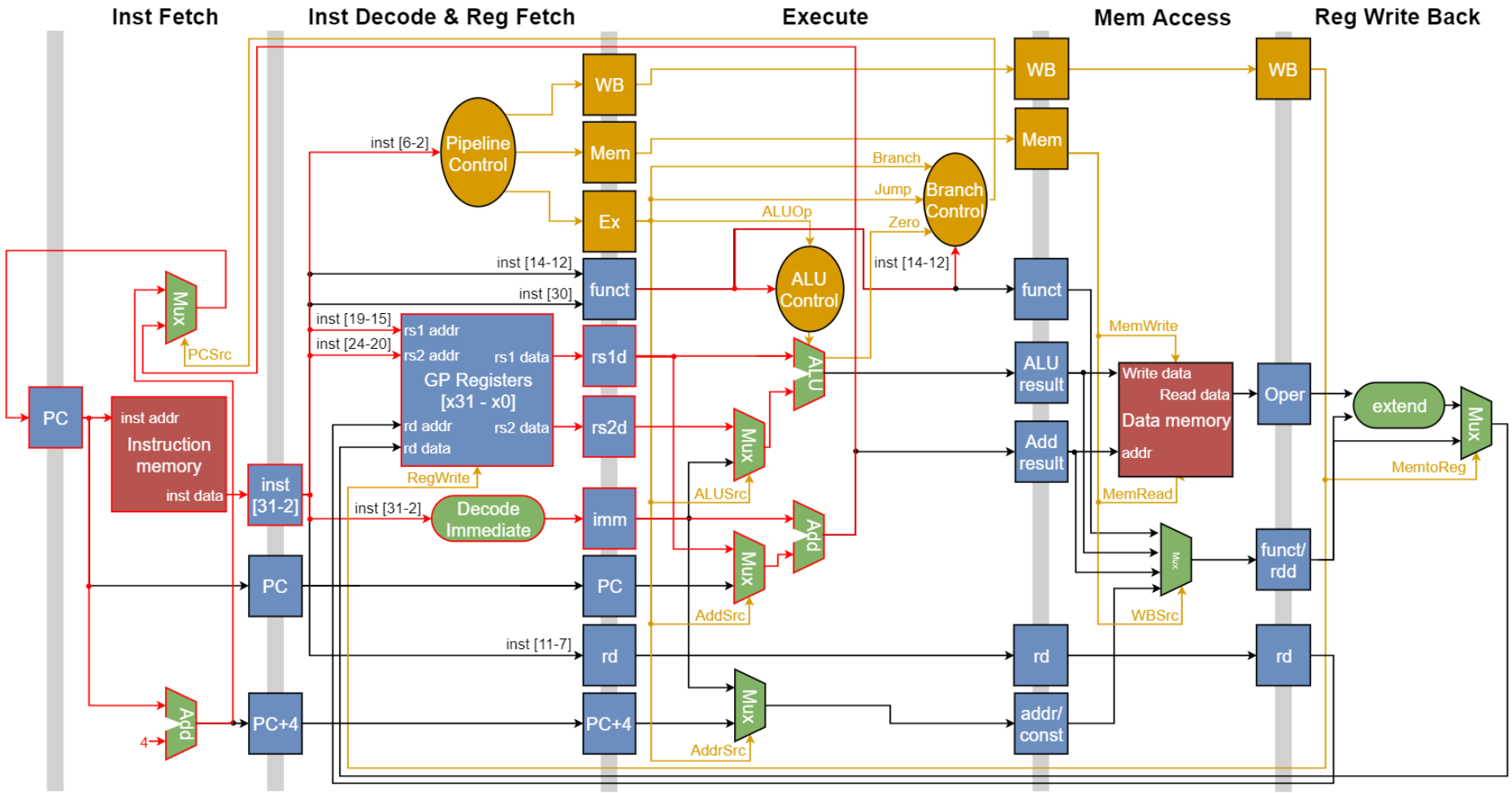
9.5 JAL Control & Data Flow



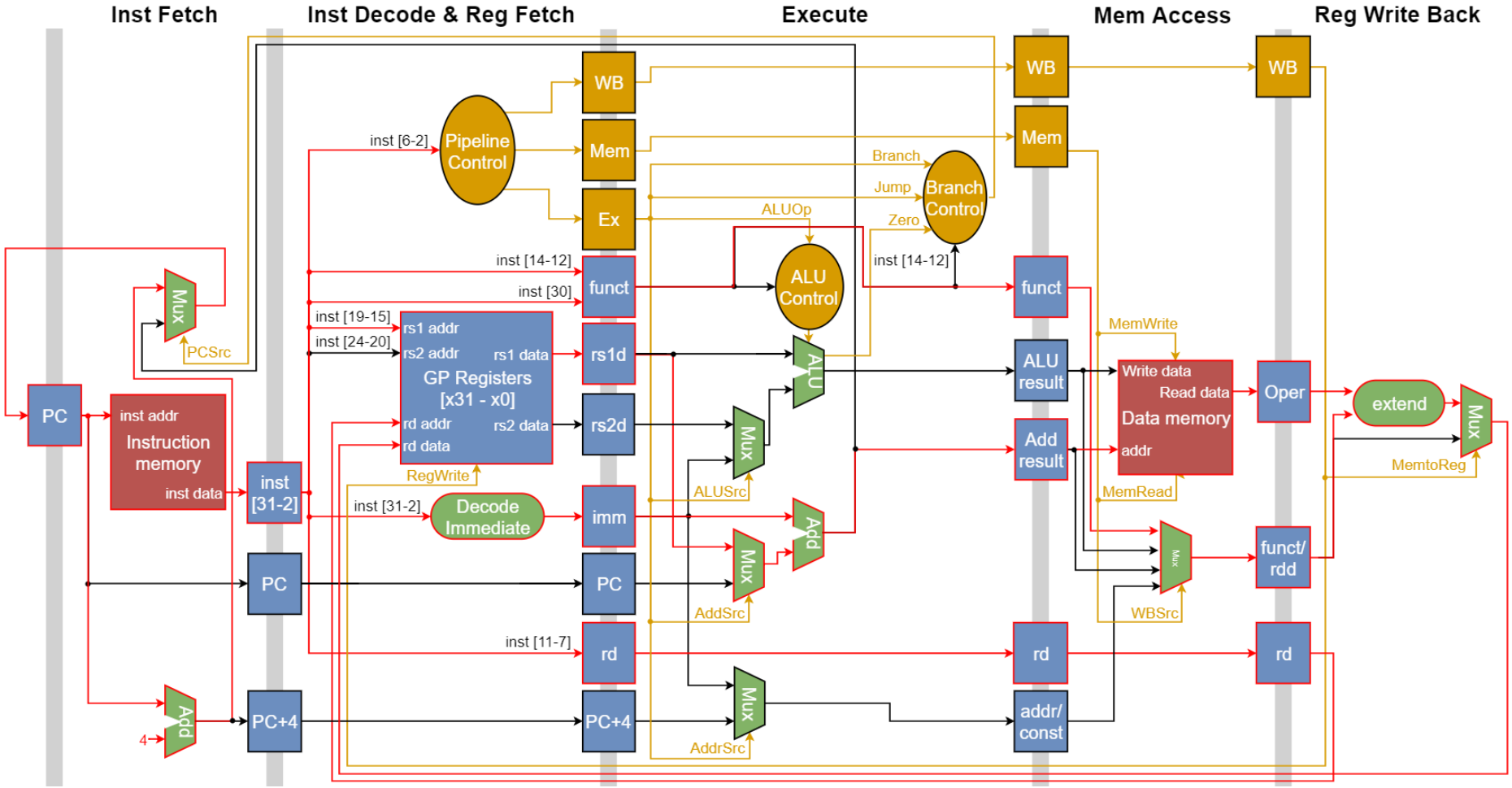
9.6 JALR Control & Data Flow



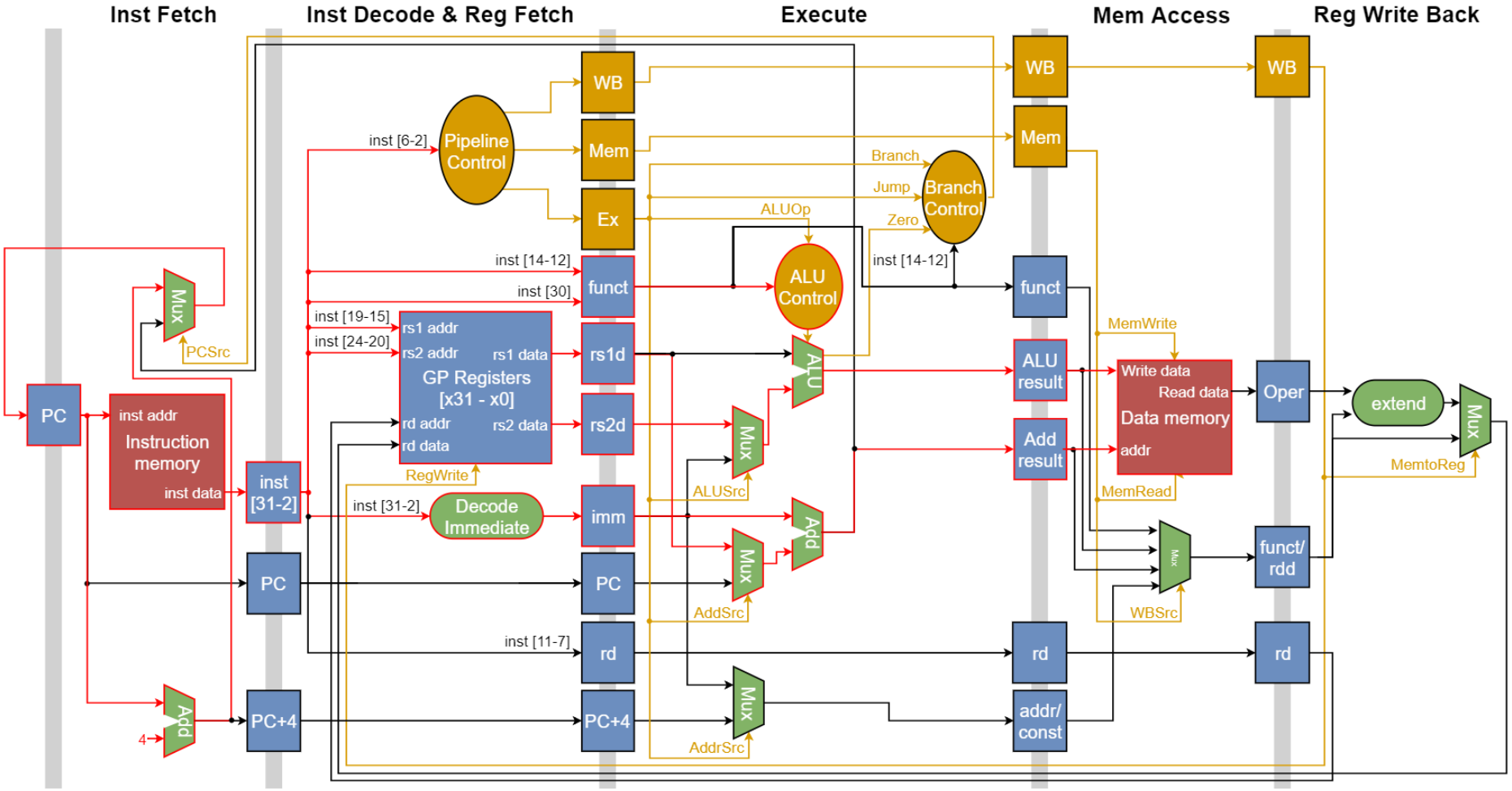
9.7 BRANCH Control & Data Flow



9.8 LOAD Control & Data Flow



9.9 STORE Control & Data Flow



10 Appendix B

10.1 U32_ALU_controller.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use work.u32_types.all;

-- define the interface between the alu controller and its external
environment
entity u32_alu_controller is
    port (
        aluop   : in std_logic_vector(1 downto 0) := (others => '0');
        funct   : in nibble_vector := (others => '0');
        alu_control : out nibble_vector := (others => '0')
    );
end u32_alu_controller;

-- define the internal organisation and operation of the alu controller
architecture rtl of u32_alu_controller is
    -- architecture declarations
    subtype aluop_vector is std_logic_vector(1 downto 0);
    subtype alu_inst_vector is std_logic_vector(5 downto 0);

    -- define aluop
    constant branch_aluop   : aluop_vector := "10";
    constant store_aluop    : aluop_vector := "11";
    constant opimm_aluop    : aluop_vector := "00";
    constant op_aluop       : aluop_vector := "01";

    -- define instructions that use the ALU
    constant beq_inst0      : alu_inst_vector := branch_aluop & "0000";
    constant beq_inst1      : alu_inst_vector := branch_aluop & "1000";
    constant bne_inst0      : alu_inst_vector := branch_aluop & "0001";
    constant bne_inst1      : alu_inst_vector := branch_aluop & "1001";
    constant blt_inst0      : alu_inst_vector := branch_aluop & "0100";
    constant blt_inst1      : alu_inst_vector := branch_aluop & "1100";
    constant bge_inst0      : alu_inst_vector := branch_aluop & "0101";
    constant bge_inst1      : alu_inst_vector := branch_aluop & "1101";
    constant bltu_inst0     : alu_inst_vector := branch_aluop & "0110";
    constant bltu_inst1     : alu_inst_vector := branch_aluop & "1110";
    constant bgeu_inst0     : alu_inst_vector := branch_aluop & "0111";
    constant bgeu_inst1     : alu_inst_vector := branch_aluop & "1111";

    constant sb_inst0       : alu_inst_vector := store_aluop & "0000";
    constant sb_inst1       : alu_inst_vector := store_aluop & "1000";
    constant sh_inst0       : alu_inst_vector := store_aluop & "0001";
    constant sh_inst1       : alu_inst_vector := store_aluop & "1001";
```

```
    constant sw_inst0       : alu_inst_vector := store_aluop & "0010";
    constant sw_inst1       : alu_inst_vector := store_aluop & "1010";

    constant addi_inst0     : alu_inst_vector := opimm_aluop & "0000";
    constant addi_inst1     : alu_inst_vector := opimm_aluop & "1000";
    constant slti_inst0     : alu_inst_vector := opimm_aluop & "0010";
    constant slti_inst1     : alu_inst_vector := opimm_aluop & "1010";
    constant sltiu_inst0    : alu_inst_vector := opimm_aluop & "0011";
    constant sltiu_inst1    : alu_inst_vector := opimm_aluop & "1011";
    constant xori_inst0     : alu_inst_vector := opimm_aluop & "0100";
    constant xori_inst1     : alu_inst_vector := opimm_aluop & "1100";
    constant ori_inst0      : alu_inst_vector := opimm_aluop & "0110";
    constant ori_inst1      : alu_inst_vector := opimm_aluop & "1110";
    constant andi_inst0     : alu_inst_vector := opimm_aluop & "0111";
    constant andi_inst1     : alu_inst_vector := opimm_aluop & "1111";
    constant slli_inst      : alu_inst_vector := opimm_aluop & "0001";
    constant srli_inst      : alu_inst_vector := opimm_aluop & "0101";
    constant srai_inst      : alu_inst_vector := opimm_aluop & "1101";

    constant add_inst       : alu_inst_vector := op_aluop & "0000";
    constant sub_inst       : alu_inst_vector := op_aluop & "1000";
    constant sll_inst       : alu_inst_vector := op_aluop & "0001";
    constant slt_inst       : alu_inst_vector := op_aluop & "0010";
    constant sltu_inst      : alu_inst_vector := op_aluop & "0011";
    constant xor_inst       : alu_inst_vector := op_aluop & "0100";
    constant srl_inst       : alu_inst_vector := op_aluop & "0101";
    constant sra_inst       : alu_inst_vector := op_aluop & "1101";
    constant or_inst        : alu_inst_vector := op_aluop & "0110";
    constant and_inst       : alu_inst_vector := op_aluop & "0111";

    signal inst_alu         : alu_inst_vector := (others => '0');

-- concurrent statements
begin
    inst_alu <= aluop & funct; -- cobined aluop and funct so with/select can
be used

    -- decode alu control signal
    with inst_alu select
    alu_control <=  alu_slt     when blt_inst1  | bge_inst1  | slti_inst1 |
blt_inst0  | bge_inst0  | slti_inst0  | slt_inst,
                    alu_sltu    when bltu_inst1 | bgeu_inst1 | sltiu_inst1 |
bltu_inst0 | bgeu_inst0 | sltiu_inst0 | sltu_inst,
                    alu_and     when andi_inst1 | andi_inst0 | and_inst,
                    alu_or      when ori_inst1  | ori_inst0 | or_inst,
                    alu_xor     when xori_inst1 | xori_inst0 | xor_inst,
                    alu_sll     when slli_inst | sll_inst,
                    alu_srl     when srli_inst | srl_inst,
```

```
                    alu_sra     when srai_inst | sra_inst,
                    alu_sub     when sub_inst | beq_inst0 | beq_inst1 |
bne_inst0 | bne_inst1,
                    alu_pass    when sb_inst1 | sh_inst1 | sw_inst1 | sb_inst0
| sh_inst0 | sw_inst0,
                    alu_add     when others;
end rtl;
```

10.2 U32_ALU.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
use ieee.numeric_std.all;
use work.u32_types.all;

-- define the interface between the alu and its external environment
entity u32_alu is
    port (
        operand1, operand2      : in word_vector := (others => '0');
        alu_control : in nibble_vector := (others => '0');
        result      : out word_vector := (others => '0');
        zero        : out std_logic := '0'
    );
end u32_alu;

-- define the internal organisation and operation of the alu
architecture rtl of u32_alu is
    -- architecture declarations
    signal shamt        : opcode_vector := (others => '0');
    signal aluresult    : word_vector := (others => '0');

-- concurrent statements
begin
    shamt <= operand2(4 downto 0);
    result <= aluresult;
    zero <= '1' when aluresult = (xlen downto 0 => '0') else '0'; -- set zero
flag high when result is 0

    process (alu_control, operand1, operand2, shamt)
    begin
        -- perform the approiate arithmatic/logic function based on ALU
control signal
        case alu_control is
            when alu_slt =>
                if operand1 < operand2 then
                    aluresult <= (xlen downto 1 => '0') & '1';
                else
                    aluresult <= (others => '0');
                end if;
            when alu_sltu =>
                if unsigned(operand1) < unsigned(operand2) then
                    aluresult <= (xlen downto 1 => '0') & '1';
                else
                    aluresult <= (others => '0');
                end if;
            when alu_and =>
```

```
                aluresult <= operand1 and operand2;
            when alu_or =>
                aluresult <= operand1 or operand2;
            when alu_xor =>
                aluresult <= operand1 xor operand2;
            when alu_sll =>
                aluresult <= std_logic_vector(shift_left(unsigned(operand1),
to_integer(unsigned(shamt))));
            when alu_srl =>
                aluresult <= std_logic_vector(shift_right(unsigned(operand1),
to_integer(unsigned(shamt))));
            when alu_sra =>
                aluresult <= std_logic_vector(shift_right(signed(operand1),
to_integer(unsigned(shamt))));
            when alu_sub =>
                aluresult <= operand1 - operand2;
            when alu_pass =>
                aluresult <= operand2;
            when others =>
                aluresult <= operand1 + operand2;
        end case;
    end process;
end rtl;
```

10.3 U32_Branch_controller.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use work.u32_types.all;

-- define the interface between the branch controller and its external
environment
entity u32_branch_controller is
    port (
        zero, branch, jump  : in std_logic := '0';
        funct               : in std_logic_vector(2 downto 0) := (others =>
'0');
        pc_src               : out std_logic := '0'
    );
end u32_branch_controller;

-- define the internal organisation and operation of the branch controller
architecture rtl of u32_branch_controller is
-- concurrent statements
begin
    -- set pc_src high when jump or branch + condition met
    pc_src <=   '1' when jump = '1' else
                '1' when branch & zero & funct = "11000" else -- BEQ and equal
                '1' when branch & zero & funct = "10001" else -- BNE and not
equal
                '1' when branch & zero & funct = "10100" else -- BLT and less
than
                '1' when branch & zero & funct = "11101" else -- BGE and
greater than or equal
                '1' when branch & zero & funct = "10110" else -- BLTU and less
than
                '1' when branch & zero & funct = "11111" else -- BGEU and
greater than or equal
                '0';
end rtl;
```

10.4 U32_Controller.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use work.u32_types.all;

-- define the interface between the pipeline controller and its external
environment
entity u32_controller is
    port(
        opcode  : in opcode_vector := (others => '0');
        control : out std_logic_vector(8 downto 0) := (others => '0')
    );
end u32_controller;

-- define the internal organisation and operation of the pipeline controller
architecture rtl of u32_controller is

begin
    -- concurrent statements
    -- decode control signals
    with opcode select
    control <=  "100001000" when lui,
                "100010000" when auipc,
                "100101100" when op,
                "100000001" when jal,
                "100000101" when jalr,
                "000011010" when branch,
                "110110100" when load,
                "001011100" when store,
                "100100000" when others; -- opimm
end rtl;
```

10.5 U32_Core.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use work.u32_types.all;

-- define the interface between the branch controller and its external
environment
entity u32_core is
    port (
        clk, write_en                              : in std_logic      :=
'0';
        write_inst, write_addr                     : in word_vector    :=
(others => '0');
        rd_data_out, addr_const_out,
        alu_result_out, add_result_out, new_pc_out  : out word_vector   :=
(others => '0');
        rd_addr_out                                : out addr_vector   :=
(others => '0');
        reg_write_out, pc_src_out                  : out std_logic     := '0'
    );
end u32_core;

-- define the internal organisation and operation of the U32 Core
architecture rtl of u32_core is
    signal  pc_src,
            clk_en,
            reg_write       : std_logic                     := '0';
    signal  new_pc,
            inst,
            pc_if_d,
            pc_d_e,
            next_pc_if_d,
            next_pc_d_e,
            rd_data,
            funct_rdd,
            rs1d,
            rs2d,
            imm,
            alu_result,
            add_result,
            addr_const,
            oper            : word_vector   := (others => '0');
    signal  control_d_e     : std_logic_vector(8 downto 0)  := (others =>
'0');
    signal  control_e_ma,
            rd_addr,
            rd_d_e,
            rd_e_ma,
```

```
            rd_ma_w         : addr_vector                   := (others =>
'0');
    signal  control_ma_w    : std_logic_vector(1 downto 0)  := (others =>
'0');
    signal  funct_d_e,
            funct_e_ma      : nibble_vector                 := (others =>
'0');
begin
    -- instantiate instruction fetch pipeline stage
    u32_inst_fetch : entity work.u32_inst_fetch
    port map (
        -- inputs
        clk => clk,
        write_en => write_en,
        pc_src => pc_src,
        write_inst => write_inst,
        write_addr => write_addr,
        new_pc => new_pc,
        -- outputs
        inst => inst,
        pc_out => pc_if_d,
        next_pc_out => next_pc_if_d,
        clk_en => clk_en
    );

    -- instantiate decode pipeline stage
    u32_decode : entity work.u32_decode
    port map (
        -- inputs
        clk => clk,
        clk_en => clk_en,
        reg_write => reg_write,
        rd_addr => rd_addr,
        inst => inst(31 downto 2),
        pc_in => pc_if_d,
        next_pc_in => next_pc_if_d,
        rd_data => rd_data,
        -- outputs
        control => control_d_e,
        funct => funct_d_e,
        rs1d => rs1d,
        rs2d => rs2d,
        imm => imm,
        pc_out => pc_d_e,
        next_pc_out => next_pc_d_e,
        rd => rd_d_e
    );
```

```
    -- instantiate execute pipeline stage
    u32_execute : entity work.u32_execute
    port map (
        -- inputs
        clk => clk,
        clk_en => clk_en,
        control => control_d_e,
        rd => rd_d_e,
        funct => funct_d_e,
        rs1d => rs1d,
        rs2d => rs2d,
        imm => imm,
        pc => pc_d_e,
        next_pc => next_pc_d_e,
        -- outputs
        control_out => control_e_ma,
        rd_out => rd_e_ma,
        funct_out => funct_e_ma,
        alu_result => alu_result,
        add_result => add_result,
        add_result_out => new_pc,
        addr_const => addr_const,
        pc_src => pc_src
    );

    -- instantiate memory access pipeline stage
    u32_mem_access : entity work.u32_mem_access
    port map (
        -- inputs
        clk => clk,
        clk_en => clk_en,
        control => control_e_ma,
        rd => rd_e_ma,
        funct => funct_e_ma,
        alu_result => alu_result,
        add_result => add_result,
        addr_const => addr_const,
        -- outputs
        oper => oper,
        funct_rdd => funct_rdd,
        rd_out => rd_ma_w,
        control_out => control_ma_w
    );

    -- instantiate writeback pipeline stage
    u32_writeback : entity work.u32_writeback
    port map (
        -- inputs
```

```
        control => control_ma_w,
        oper => oper,
        funct_rdd => funct_rdd,
        rd => rd_ma_w,
        -- outputs
        rd_out => rd_addr,
        rd_data => rd_data,
        reg_write => reg_write
    );


    addr_const_out <= addr_const;
    alu_result_out <= alu_result;
    add_result_out <= add_result;
    new_pc_out <= new_pc;
    pc_src_out <= pc_src;
    rd_data_out <= rd_data;
    rd_addr_out <= rd_addr;
    reg_write_out <= reg_write;
end rtl;
```

10.6 U32_Data_Extender.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use work.u32_types.all;

-- define the interface between the data extender and its external environment
entity u32_data_extender is
    port (
        funct       : in std_logic_vector(2 downto 0)   := (others => '0');
        oper        : in word_vector                    := (others => '0');
        oper_out    : out word_vector                   := (others => '0')
    );
end u32_data_extender;

-- define the internal organisation and operation of the data extender
architecture rtl of u32_data_extender is
    constant lb   : std_logic_vector(2 downto 0) := "000";
    constant lh   : std_logic_vector(2 downto 0) := "001";
    constant lbu   : std_logic_vector(2 downto 0) := "100";
    constant lhu   : std_logic_vector(2 downto 0) := "101";
begin
    with funct select
    oper_out <= (xlen downto 7 => oper(7)) & oper(6 downto 0) when lb, -- sign
extend byte
                (xlen downto 15 => oper(15)) & oper(14 downto 0) when lh, --
sign extend half word
                (xlen downto 8 => '0') & oper(7 downto 0) when lbu, -- zero
extend byte
                (xlen downto 16 => '0') & oper(15 downto 0) when lhu, -- zero
extend half word
                oper when others; -- passthrough
end rtl;
```

10.7 U32_Data_Memory.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.u32_types.all;

-- define the interface between the data memory and its external environment
entity u32_data_memory is
    port (
        clk, write_en, read_en  : in std_logic := '0';
        funct                   : in std_logic_vector(1 downto 0) := (others
=> '0');
        addr, write_data        : in word_vector := (others => '0');
        read_data               : out word_vector := (others => '0')
    );
end u32_data_memory;

-- define the internal organisation and operation of the data memory
architecture behavioral of u32_data_memory is
    -- define array
    type byte_ram is array (0 to (data_mem_size - 1)) of byte_vector;

    subtype byte_addr is natural range 0 to (data_mem_size - 1);

    signal ram_addr : byte_addr := 0;
    signal data_mem : byte_ram := (others => (others => '0'));
    signal data     : word_vector := (others => '0');

    constant third_last_byte    : byte_addr := (data_mem_size - 3);
    constant second_last_byte   : byte_addr := (data_mem_size - 2);
    constant last_byte          : byte_addr := (data_mem_size - 1);

    constant byte           : std_logic_vector(1 downto 0) := "00";
    constant half_word      : std_logic_vector(1 downto 0) := "01";
    constant word           : std_logic_vector(1 downto 0) := "10";
begin
    ram_addr <= to_integer(unsigned(addr((log2(data_mem_size) - 1) downto
0)));

    -- read word from data memory, padding with zeros if the address is the
last, second to last or third to
    -- last byte address
    data <= x"000000"   & data_mem(ram_addr)       when ram_addr = last_byte
else
            x"0000"     & data_mem(ram_addr + 1)
                        & data_mem(ram_addr)       when ram_addr =
second_last_byte else
            x"00"       & data_mem(ram_addr + 2)
```

```
                        & data_mem(ram_addr + 1)
                        & data_mem(ram_addr)       when ram_addr =
third_last_byte else
            data_mem(ram_addr + 3) &
            data_mem(ram_addr + 2) &
            data_mem(ram_addr + 1) &
            data_mem(ram_addr);

    read_data <= data when (read_en = '1') else x"00000000"; -- do not return
data unless read enable is high

    process begin
        wait until rising_edge(clk);
        -- write word into data memory when address is anything other than the
last, second to last or
        -- third to last byte address
        if (write_en = '1' and funct = word and ram_addr /= last_byte and
ram_addr /= second_last_byte and
            ram_addr /= third_last_byte) then
            data_mem(ram_addr) <= write_data(7 downto 0);
            data_mem(ram_addr + 1) <= write_data(15 downto 8);
            data_mem(ram_addr + 2) <= write_data(23 downto 16);
            data_mem(ram_addr + 3) <= write_data(31 downto 24);
        -- write first three bytes of word into data mem when address is third
to last byte address
        elsif (write_en = '1' and funct = word and ram_addr /= last_byte and
ram_addr /= second_last_byte) then
            data_mem(ram_addr) <= write_data(7 downto 0);
            data_mem(ram_addr + 1) <= write_data(15 downto 8);
            data_mem(ram_addr + 2) <= write_data(23 downto 16);
        -- write half-word into data mem or last two byte of a word when
address is second to last byte address
        elsif (write_en = '1' and funct /= byte and ram_addr /= last_byte)
then
            data_mem(ram_addr) <= write_data(7 downto 0);
            data_mem(ram_addr + 1) <= write_data(15 downto 8);
        -- write byte into data memory
        elsif (write_en = '1') then
            data_mem(ram_addr) <= write_data(7 downto 0);
        end if;
    end process;
end behavioral;
```

10.8 U32_Decode.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use work.u32_types.all;

-- define the interface between the decode pipeline stage and its external
environment
entity u32_decode is
    port(
        clk, clk_en, reg_write                  : in std_logic
:= '0';
        rd_addr                                 : in addr_vector
:= (others => '0');
        inst                                    : in inst_vector
:= (others => '0');
        pc_in, next_pc_in, rd_data              : in word_vector
:= (others => '0');
        control                                 : out std_logic_vector(8
downto 0)  := (others => '0');
        funct                                   : out nibble_vector
:= (others => '0');
        rs1d, rs2d, imm, pc_out, next_pc_out    : out word_vector
:= (others => '0');
        rd                                      : out addr_vector
:= (others => '0')
    );
end u32_decode;

-- define the internal organisation and operation of the decode pipeline stage
architecture rtl of u32_decode is
    signal imm_reg      : word_vector                   := (others => '0');
    signal rs1d_reg     : word_vector                   := (others => '0');
    signal rs2d_reg     : word_vector                   := (others => '0');
    signal control_reg  : std_logic_vector(8 downto 0)  := (others => '0');
begin
    -- concurrent statements
    -- instantiate controller
    u32_controller : entity work.u32_controller
    port map (
        opcode => inst(6 downto 2),
        control => control_reg
    );

    -- instantiate general purpose registers
    u32_gp_registers : entity work.u32_gp_registers
    port map (
        clk => clk,
        reg_write => reg_write,
```

```
        rs1_addr => inst(19 downto 15),
        rs2_addr => inst(24 downto 20),
        rd_addr => rd_addr,
        rd_data => rd_data,
        rs1_data => rs1d_reg,
        rs2_data => rs2d_reg
    );

    -- instantiate immediate decoder
    u32_immediate_decoder : entity work.u32_immediate_decoder
    port map (
        inst => inst,
        imm => imm_reg
    );

    -- sequential statements
    process begin
        wait until falling_edge(clk);
        -- pipeline registers
        if clk_en = '1' then
            control <= control_reg;
            funct <= inst(30) & inst(14 downto 12);
            rs1d <= rs1d_reg;
            rs2d <= rs2d_reg;
            imm <= imm_reg;
            pc_out <= pc_in;
            rd <= inst(11 downto 7);
            next_pc_out <= next_pc_in;
        end if;
    end process;
end rtl;
```

10.9 U32_Execute.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
use work.u32_types.all;

-- define the interface between the execute pipeline stage and its external
environment
entity u32_execute is
    port(
        clk, clk_en                     : in std_logic                      :=
'0';
        control                         : in std_logic_vector(8 downto 0)   :=
(others => '0');
        rd                              : in addr_vector                    :=
(others => '0');
        funct                           : in nibble_vector                  :=
(others => '0');
        rs1d, rs2d, imm, pc, next_pc    : in word_vector                    :=
(others => '0');
        control_out, rd_out             : out addr_vector                   :=
(others => '0');
        funct_out                       : out nibble_vector                 :=
(others => '0');
        alu_result, add_result,
        add_result_out, addr_const      : out word_vector                   :=
(others => '0');
        pc_src                           : out std_logic
:= '0'
    );
end u32_execute;

-- define the internal organisation and operation of the execute pipeline
stage
architecture rtl of u32_execute is
    signal jump, branch, addsrc_aluop0,
            addrsrc_alusrc, aluop1, zero  : std_logic     := '0';

    signal operand2, result, add_result_reg,
            addr_const_reg, addoperand2         : word_vector    := (others =>
'0');
    signal alu_control                          : nibble_vector := (others =>
'0');
begin
    -- concurrent statements
    jump <= control(0);
    branch <= control(1);
    addsrc_aluop0 <= control(2);
```

```
    addrsrc_alusrc <= control(3);
    aluop1 <= control(4);

    -- connect add_result directly to output pin. Perminalteley set bit 0 to 0
as jump and branch
    -- should only increment the proghram counter in multiples of two bytes
    add_result_out <= add_result_reg(xlen downto 1) & '0';

    -- control source of operands into ALU, Add unit and addr/const pipeline
register (multiplexers)
    operand2 <= rs2d when addrsrc_alusrc = '1' else imm;
    addoperand2 <= rs1d when addsrc_aluop0 = '1' else pc;
    addr_const_reg <= imm when addrsrc_alusrc = '1' else next_pc;

    -- Add unit
    add_result_reg <= imm + addoperand2;

    -- instantiate ALU
    u32_alu : entity work.u32_alu
    port map (
        operand1 => rs1d,
        operand2 => operand2,
        alu_control => alu_control,
        result => result,
        zero => zero
    );

    -- instantiate ALU controller
    u32_alu_controller : entity work.u32_alu_controller
    port map (
        aluop => aluop1 & addsrc_aluop0,
        funct => funct,
        alu_control => alu_control
    );

    -- instantiate Branch controller
    u32_branch_controller : entity work.u32_branch_controller
    port map (
        jump => jump,
        branch => branch,
        zero => zero,
        funct => funct(2 downto 0),
        pc_src => pc_src
    );

    -- sequential statements
    process begin
        wait until falling_edge(clk);
```

```
        if clk_en = '1' then
            control_out <= control(8 downto 4);
            funct_out <= funct;
            alu_result <= result;
            add_result <= add_result_reg;
            rd_out <= rd;
            addr_const <= addr_const_reg;
        end if;
    end process;
end rtl;
```

10.10 U32_GP_Registers.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.u32_types.all;

-- define the interface between the general-purpose registers and its external
environment
entity u32_gp_registers is
    port(
        clk, reg_write              : in std_logic := '0';
        rs1_addr, rs2_addr, rd_addr : in opcode_vector := (others => '0');
        rd_data                     : in word_vector := (others => '0');
        rs1_data, rs2_data          : out word_vector := (others => '0')
    );
end u32_gp_registers;

-- define the internal organisation and operation of the general-purpose
registers
architecture rtl of u32_gp_registers is
    type word_matrix is array (0 to 31) of word_vector;
    signal gp_registers : word_matrix := (others => (others => '0'));
begin
    -- concurrent statements (read)
    rs1_data <= gp_registers(to_integer(unsigned(rs1_addr)));
    rs2_data <= gp_registers(to_integer(unsigned(rs2_addr)));
    -- sequential statements (write)
    process begin
        wait until rising_edge(clk);
        if (reg_write = '1' and rd_addr /= "00000") then
            gp_registers(to_integer(unsigned(rd_addr))) <= rd_data;
        end if;
    end process;
end rtl;
```

10.11 U32_Immediate_Decoder.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use work.u32_types.all;

-- define the interface between the immediate decoder and its external
environment
entity u32_immediate_decoder is
    port (
        inst    : in inst_vector := (others => '0');
        imm     : out word_vector := (others => '0')
    );
end u32_immediate_decoder;

-- define the internal organisation and operation of the immediate decoder
architecture rtl of u32_immediate_decoder is
    signal opcode : opcode_vector := (others => '0');
begin
    opcode <= inst(6 downto 2); -- isolate opcode from instrcution

    -- decode immediate
    with opcode select
    imm <=  inst(xlen downto 12) & (11 downto 0 => '0')
when lui | auipc, -- U-type
            (xlen downto 20 => inst(xlen)) & inst(19 downto 12) & inst(20) &
inst(30 downto 21) & '0'   when jal, -- J-type
            (xlen downto 12 => inst(xlen)) & inst(7) & inst(30 downto 25) &
inst(11 downto 8) & '0'     when branch, -- B-type
            (xlen downto 11 => inst(xlen)) & inst(30 downto 25) & inst(11
downto 7)                     when store, -- S-type
            (xlen downto 11 => inst(xlen)) & inst(30 downto 20)
when others; -- I-type
end rtl;
```

10.12 U32_Inst_Fetch.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;
use work.u32_types.all;

-- define the interface between the instruction fetch pipeline stage and its
external environment
entity u32_inst_fetch is
    port (
        clk, write_en, pc_src           : in std_logic      := '0';
        write_inst, write_addr, new_pc  : in word_vector    := (others =>
'0');
        inst, pc_out, next_pc_out       : out word_vector   := (others =>
'0');
        clk_en                          : out std_logic     := '0'
    );
end u32_inst_fetch;

-- define the internal organisation and operation of the instruction fetch
pipeline stage
architecture rtl of u32_inst_fetch is
    constant increment      : std_logic_vector(xlen downto 0)   := ((xlen
downto 3 => '0') & "100");
    constant inst_mem_xlen  : natural                           :=
((inst_mem_size / 4) - 1);

    type ram is array (0 to inst_mem_xlen) of word_vector;

    signal next_pc, pc, inst_reg            : word_vector   := (others =>
'0');
    signal inst_mem                         : ram           := (others =>
(others => '0'));
    signal true_write_addr, true_read_addr  : natural       := 0;
begin
    -- concurrent statements
    -- convert byte address into word address
    true_read_addr <= to_integer(unsigned(pc)) / 4;
    true_write_addr <= to_integer(unsigned(write_addr)) / 4;

    -- read instruction out of instruction memory
    inst_reg <= inst_mem(true_read_addr);

    -- update program counter
    next_pc <=    new_pc when pc_src = '1' else
                  pc + increment;
```

```
    -- sequential statements
    process (clk)
        variable count  : natural range 0 to inst_mem_xlen  := 0;
    begin
        if rising_edge(clk) then
            -- stall pipeline until instruction memory filled
            if count = inst_mem_xlen then
                clk_en <= '1';
            else
                count := count + 1;
            end if;

            -- write instruction into instruction memory
            if write_en = '1' then
                inst_mem(true_write_addr) <= write_inst;
            end if;
        elsif falling_edge(clk) then
            -- pipeline registers
            if clk_en = '1' then
                pc <= next_pc;
                pc_out <= pc;
                next_pc_out <= pc + increment;
                inst <= inst_reg;
            end if;
        end if;
    end process;
end rtl;
```

10.13 U32_Mem_Access.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use work.u32_types.all;

-- define the interface between the memory access pipeline stage and its
external environment
entity u32_mem_access is
    port (
        clk, clk_en                         : in std_logic
:= '0';
        control, rd                         : in std_logic_vector(4 downto 0)
:= (others => '0');
        funct                               : in nibble_vector
:= (others => '0');
        alu_result, add_result, addr_const  : in word_vector
:= (others => '0');
        oper, funct_rdd                     : out word_vector
:= (others => '0');
        rd_out                              : out addr_vector
:= (others => '0');
        control_out                         : out std_logic_vector(1 downto 0)
:= (others => '0')
    );
end u32_mem_access;

-- define the internal organisation and operation of the memory access
pipeline stage
architecture rtl of u32_mem_access is
    signal wbsrc                    : std_logic_vector(1 downto 0)  := (others
=> '0');
    signal write_en, read_en        : std_logic                     := '0';
    signal funct_rdd_reg, oper_reg  : word_vector                   := (others
=> '0');
begin
    -- concurrent statements
    wbsrc <= control(1 downto 0);
    write_en <= control(2);
    read_en <= control(3);

    -- control source of data into funct/rdd pipeline register (multiplexer)
    with wbsrc select
    funct_rdd_reg <=    addr_const when "00",
                        add_result when "01",
                        alu_result when "10",
                        (31 downto 4 => '0') & funct when others;

    -- instantiate data memory
```

```
    u32_data_memory : entity work.u32_data_memory
    port map (
        clk => clk,
        write_en => write_en,
        read_en => read_en,
        funct => funct(1 downto 0),
        addr => add_result,
        write_data => alu_result,
        read_data => oper_reg
    );

    -- sequential statements
    process begin
        wait until falling_edge(clk);
        if clk_en = '1' then
            control_out <= control(4 downto 3);
            funct_rdd <= funct_rdd_reg;
            oper <= oper_reg;
            rd_out <= rd;
        end if;
    end process;
end rtl;
```

10.14 U32_types.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

package u32_types is
    -- set bit length of architecture
    constant xlen           : natural := 31;

    -- opcode suffix, always 11 for RV32I architectures
    constant suffix : std_logic_vector(1 downto 0) := "11";

    -- size of ram in bytes (should be a power of 2 e.g. 256, 512, 1024, etc)
    constant data_mem_size : natural := 256;
    constant inst_mem_size : natural := 256;

    -- subtypes used throughout the u32 core
    subtype word_vector is std_logic_vector(31 downto 0);
    subtype half_word_vector is std_logic_vector(15 downto 0);
    subtype byte_vector is std_logic_vector(7 downto 0);
    subtype nibble_vector is std_logic_vector(3 downto 0);
    subtype inst_vector is std_logic_vector(31 downto 2);
    subtype opcode_vector is std_logic_vector(6 downto 2);
    subtype addr_vector is std_logic_vector(4 downto 0);

    -- opcodes reduced down from 7 bits to 5 by excluding the architectures
suffix
    constant lui    : opcode_vector := "01101";
    constant auipc  : opcode_vector := "00101";
    constant opimm  : opcode_vector := "00100";
    constant op     : opcode_vector := "01100";
    constant jal    : opcode_vector := "11011";
    constant jalr   : opcode_vector := "11001";
    constant branch : opcode_vector := "11000";
    constant load   : opcode_vector := "00000";
    constant store  : opcode_vector := "01000";

    -- alu cotnrol
    constant alu_add    : nibble_vector := "0000";
    constant alu_slt    : nibble_vector := "0001";
    constant alu_sltu   : nibble_vector := "0010";
    constant alu_and    : nibble_vector := "0011";
    constant alu_or     : nibble_vector := "0100";
    constant alu_xor    : nibble_vector := "0101";
    constant alu_sll    : nibble_vector := "0110";
    constant alu_srl    : nibble_vector := "0111";
    constant alu_sra    : nibble_vector := "1000";
    constant alu_sub    : nibble_vector := "1001";
```

```
    constant alu_pass   : nibble_vector := "1010";

    function log2 (x : natural) return natural;
end package u32_types;

package body u32_types is
    function log2 (x : natural) return natural is
        variable i : natural := 0;
    begin
        i := 0;
        while (2**i < x) and i < 31 loop
            i := i + 1;
        end loop;
        return i;
    end function;
end package body u32_types;
```

10.15 U32_Writeback.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use work.u32_types.all;

-- define the interface between the write back pipeline stage and its external
environment
entity u32_writeback is
    port(
        control             : in std_logic_vector(1 downto 0)   := (others =>
'0');
        oper, funct_rdd     : in word_vector                    := (others =>
'0');
        rd                  : in addr_vector                    := (others =>
'0');
        rd_out              : out addr_vector                   := (others =>
'0');
        rd_data             : out word_vector                   := (others =>
'0');
        reg_write           : out std_logic                     := '0'
    );
end u32_writeback;

-- define the internal organisation and operation of the write back pipeline
stage
architecture rtl of u32_writeback is
    signal oper_out : word_vector := (others => '0');
begin
    -- concurrent statements
    u32_data_extender : entity work.u32_data_extender
    port map (
        funct => funct_rdd(2 downto 0),
        oper => oper,
        oper_out => oper_out
    );

    rd_data <= oper_out when control(0) else funct_rdd;
    reg_write <= control(1);
    rd_out <= rd;
end rtl;
```

11 Appendix C

11.1 ALU_Controller_testbench.vhd

```
library ieee;
use ieee.std_logic_1164.all;

entity alu_controller_testbench is
end alu_controller_testbench;

-- define the internal organisation and operation of the alu controller
testbench
architecture behaviour of alu_controller_testbench is
    -- architecture declarations
    constant time_delta : time := 100 ns;

    signal aluop                : std_logic_vector(1 downto 0) := (others =>
'0');
    signal funct, alu_control   : std_logic_vector(3 downto 0) := (others =>
'0');
-- concurrent statements
begin
    -- instantiate alu_controller
    u32_alu_controller : entity work.u32_alu_controller
    port map (
        aluop => aluop,
        funct => funct,
        alu_control => alu_control
    );

    process
        procedure test(
            constant alu_operation  : in std_logic_vector(1 downto 0);
            constant alu_function   : in std_logic_vector(3 downto 0);
            constant expected       : in std_logic_vector(3 downto 0)
        ) is

        begin
            -- assign values to circuit inputs
            aluop <= alu_operation;
            funct <= alu_function;

            wait for time_delta;

            assert alu_control = expected
            report "Unexpected result: " &
            "operation = " & to_string(alu_operation) & "; " &
            "function = " & to_string(alu_function) & "; " &
            "result = " & to_string(alu_control) & "; " &
```

```
            "expected = " & to_string(expected)
            severity error;
        end procedure test;
    begin
        -- ADD
        test("10", "0000", "0000"); -- BEQ
        test("10", "1000", "0000"); -- BEQ
        test("10", "0001", "0000"); -- BNE
        test("10", "1001", "0000"); -- BNE
        test("00", "0000", "0000"); -- ADDI
        test("00", "1000", "0000"); -- ADDI
        test("01", "0000", "0000"); -- ADD

        -- SLT
        test("10", "0100", "0001"); -- BLT
        test("10", "1100", "0001"); -- BLT
        test("10", "0101", "0001"); -- BGE
        test("10", "1101", "0001"); -- BGE
        test("00", "0010", "0001"); -- SLTI
        test("00", "1010", "0001"); -- SLTI
        test("01", "0010", "0001"); -- SLT

        -- SLTU
        test("10", "0110", "0010"); -- BLTU
        test("10", "1110", "0010"); -- BLTU
        test("10", "0111", "0010"); -- BGEU
        test("10", "1111", "0010"); -- BGEU
        test("00", "0011", "0010"); -- SLTIU
        test("00", "1011", "0010"); -- SLTIU
        test("01", "0011", "0010"); -- SLTU

        -- PASS
        test("11", "0000", "1010"); -- SB
        test("11", "1000", "1010"); -- SB
        test("11", "0001", "1010"); -- SH
        test("11", "1001", "1010"); -- SH
        test("11", "0010", "1010"); -- SW
        test("11", "1010", "1010"); -- SW

        -- XOR
        test("00", "0100", "0101"); -- XORI
        test("00", "1100", "0101"); -- XORI
        test("01", "0100", "0101"); -- XOR

        -- OR
        test("00", "0110", "0100"); -- ORI
        test("00", "1110", "0100"); -- ORI
        test("01", "0110", "0100"); -- OR
```

```
        -- AND
        test("00", "0111", "0011"); -- ANDI
        test("00", "1111", "0011"); -- ANDI
        test("01", "0111", "0011"); -- AND

        -- SLL
        test("00", "0001", "0110"); -- SLLI
        test("01", "0001", "0110"); -- SLL

        -- SRL
        test("00", "0101", "0111"); -- SRLI
        test("01", "0101", "0111"); -- SRL

        -- SRA
        test("00", "1101", "1000"); -- SRAI
        test("01", "1101", "1000"); -- SRA

        -- SUB
        test("01", "1000", "1001"); -- SUB
        wait;
    end process;
end behaviour;
```

11.2 ALU_testbench.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_signed.all;

entity alu_testbench is
end alu_testbench;

-- define the internal organisation and operation of the alu testbench
architecture behaviour of alu_testbench is
    -- architecture declarations
    constant time_delta : time := 100 ns;

    signal operand1, operand2   : std_logic_vector(31 downto 0) := (others =>
'0');
    signal alu_control          : std_logic_vector(3 downto 0) := (others =>
'0');
    signal result               : std_logic_vector(31 downto 0) := (others =>
'0');
    signal zero                 : std_logic := '0';

-- concurrent statements
begin
    -- instantiate alu
    u32_alu : entity work.u32_alu
    port map (
        operand1 => operand1,
        operand2 => operand2,
        alu_control => alu_control,
        result => result,
        zero => zero
    );

    process
        procedure test(
            constant operation  : in string;
            constant value1     : in integer;
            constant value2     : in integer;
            constant expected   : in integer
        ) is
            variable res : integer;
        begin
            -- assign values to circuit inputs
            operand1 <= std_logic_vector(to_signed(value1, operand1'length));
            operand2 <= std_logic_vector(to_signed(value2, operand2'length));

            wait for time_delta;
```

```
        res := conv_integer(result);
        assert res = expected
        report "Unexpected result: " &
        "operation = " & operation & "; " &
        "operand1 = " & integer'image(value1) & "; " &
        "operand2 = " & integer'image(value2) & "; " &
        "result = " & integer'image(res) & "; " &
        "expected = " & integer'image(expected)
        severity error;
    end procedure test;

    procedure test_add(
        constant value1     : in integer;
        constant value2     : in integer;
        constant expected   : in integer
    ) is
        constant operation : string := "ADD";
    begin
        alu_control <= "0000";
        test(operation, value1, value2, expected);
    end procedure test_add;

    procedure test_slt(
        constant value1     : in integer;
        constant value2     : in integer;
        constant expected   : in integer
    ) is
        constant operation : string := "SLT";
    begin
        alu_control <= "0001";
        test(operation, value1, value2, expected);
    end procedure test_slt;

    procedure test_sltu(
        constant value1     : in integer;
        constant value2     : in integer;
        constant expected   : in integer
    ) is
        constant operation : string := "SLTU";
    begin
        alu_control <= "0010";
        test(operation, value1, value2, expected);
    end procedure test_sltu;

    procedure test_and(
        constant value1     : in integer;
        constant value2     : in integer;
```

```
        constant expected   : in integer
    ) is
        constant operation : string := "AND";
    begin
        alu_control <= "0011";
        test(operation, value1, value2, expected);
    end procedure test_and;

    procedure test_or(
        constant value1     : in integer;
        constant value2     : in integer;
        constant expected   : in integer
    ) is
        constant operation : string := "OR";
    begin
        alu_control <= "0100";
        test(operation, value1, value2, expected);
    end procedure test_or;

    procedure test_xor(
        constant value1     : in integer;
        constant value2     : in integer;
        constant expected   : in integer
    ) is
        constant operation : string := "XOR";
    begin
        alu_control <= "0101";
        test(operation, value1, value2, expected);
    end procedure test_xor;

    procedure test_sll(
        constant value1     : in integer;
        constant value2     : in integer;
        constant expected   : in integer
    ) is
        constant operation : string := "SLL";
    begin
        alu_control <= "0110";
        test(operation, value1, value2, expected);
    end procedure test_sll;

    procedure test_srl(
        constant value1     : in integer;
        constant value2     : in integer;
        constant expected   : in integer
    ) is
        constant operation : string := "SRL";
    begin
```

```
            alu_control <= "0111";
            test(operation, value1, value2, expected);
        end procedure test_srl;

        procedure test_sra(
            constant value1     : in integer;
            constant value2     : in integer;
            constant expected   : in integer
        ) is
            constant operation : string := "SRA";
        begin
            alu_control <= "1000";
            test(operation, value1, value2, expected);
        end procedure test_sra;

        procedure test_sub(
            constant value1     : in integer;
            constant value2     : in integer;
            constant expected   : in integer
        ) is
            constant operation : string := "SUB";
        begin
            alu_control <= "1001";
            test(operation, value1, value2, expected);
        end procedure test_sub;

        procedure test_pass(
            constant value1     : in integer;
            constant value2     : in integer;
            constant expected   : in integer
        ) is
            constant operation : string := "PASS";
        begin
            alu_control <= "1010";
            test(operation, value1, value2, expected);
        end procedure test_pass;
    begin
        -- (operand1, operand2, expected)
        test_add(0, 0, 0);
        test_add(10, 10, 20);
        test_add(-10, -10, -20);
        test_add(-10, 10, 0);
        test_add(2147483647, -2147483648, -1);
        test_add(-2147483648, -1, 2147483647); -- overflow
        test_add(2147483647, 1, -2147483648); -- overflow

        test_slt(-1, 0, 1);
        test_slt(0, -1, 0);
```

```
        test_slt(-2147483648, 2147483647, 1);
        test_slt(2147483647, -2147483648, 0);
        test_slt(0, 0, 0);

        test_sltu(-1, 0, 0);
        test_sltu(0, -1, 1);
        test_sltu(-2147483648, 2147483647, 0);
        test_sltu(2147483647, -2147483648, 1);
        test_sltu(0, 0, 0);

        test_and(9, 13, 9);
        test_and(-1, -1, -1); -- all 1's
        test_and(0, 0, 0); -- all 0's
        test_and(-1, 0, 0);
        test_and(0, -1, 0);

        test_or(9, 13, 13);
        test_or(-1, -1, -1); -- all 1's
        test_or(0, 0, 0); -- all 0's
        test_or(-1, 0, -1);
        test_or(0, -1, -1);

        test_xor(9, 13, 4);
        test_xor(-1, -1, 0);
        test_xor(0, 0, 0);
        test_xor(-1, 0, -1);
        test_xor(0, -1, -1);

        test_sll(5, 10, 5120);
        test_sll(-1, -1, -2147483648);
        test_sll(-1, 31, -2147483648);
        test_sll(0, 0, 0);
        test_sll(1431655765, 1, -1431655766);

        test_srl(-65538, 15, 131069);
        test_srl(-1, -1, 1);
        test_srl(-1, 31, 1);
        test_srl(0, 0, 0);
        test_srl(-1431655766, 1, 1431655765);

        test_sra(-1431655766, 1, -715827883);
        test_sra(-1, -1, -1);
        test_sra(-1, 31, -1);
        test_sra(0, 0, 0);
        test_sra(-715827883, 1, -357913942);

        test_sub(0, 0, 0);
        test_sub(10, 10, 0);
```

```
        test_sub(-10, -10, 0);
        test_sub(-10, 10, -20);
        test_sub(-2147483648, -2147483648, 0);
        test_sub(2147483647, 2147483647, 0);
        test_sub(2147483647, -1, -2147483648); -- overflow
        test_sub(-2147483648, 1, 2147483647); -- overflow

        test_pass(-100, 8, 8);
        wait;
    end process;
end behaviour;
```

11.3 Controller_testbench.vhd

```
library ieee;
use ieee.std_logic_1164.all;

entity controller_testbench is
end controller_testbench;

-- define the internal organisation and operation of the pipeline controller
testbench
architecture behaviour of controller_testbench is
    -- architecture declarations
    constant time_delta : time := 100 ns;

    signal opcode   : std_logic_vector(4 downto 0) := (others => '0');
    signal control  : std_logic_vector(8 downto 0) := (others => '0');
-- concurrent statements
begin
    -- instantiate controller
    controller : entity work.u32_controller
    port map (
        opcode => opcode,
        control => control
    );

    process
        procedure test(
            constant operation  : in std_logic_vector(4 downto 0);
            constant expected       : in std_logic_vector(8 downto 0)
        ) is

        begin
            -- assign values to circuit inputs
            opcode <= operation;

            -- wait for controller to respond
            wait for time_delta;

            -- log any unexpted outputs
            assert control = expected
            report "Unexpected result: " &
            "operation = " & to_string(operation) & "; " &
            "result = " & to_string(control) & "; " &
            "expected = " & to_string(expected)
            severity error;
        end procedure test;
    begin
        test("01101", "100001000"); -- LUI
        test("00101", "100010000"); -- AUIPC
```

```
        test("11011", "100000001"); -- JAL
        test("11001", "100000101"); -- JALR
        test("11000", "000011010"); -- BRANCH
        test("00000", "110110100"); -- LOAD
        test("01000", "001011100"); -- STORE
        test("00100", "100100000"); -- OP-IMM
        test("01100", "100101100"); -- OP
        wait;
    end process;
end behaviour;
```

11.4 Core_testbench.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use work.getto_compiler.all;

entity core_testbench is
end core_testbench;

-- define the internal organisation and operation of the core testbench
architecture behaviour of core_testbench is
    -- architecture declarations
    signal clk, write_en, reg_write, pc_src : std_logic                    :=
'0';
    signal write_inst, write_addr, rd_data,
            addr_const, alu_result,
            add_result, new_pc              : std_logic_vector(31 downto 0) :=
(others => '0');
    signal rd_addr                          : std_logic_vector(4 downto 0)  :=
(others => '0');
    signal signals                          : std_logic_vector(65 downto 0) :=
(others => '0');
-- concurrent statements
begin
    -- instantiate core
    core : entity work.u32_core
    port map (
        -- inputs
        clk => clk,
        write_en => write_en,
        write_inst => write_inst,
        write_addr => write_addr,
        -- outputs
        addr_const_out => addr_const,
        alu_result_out => alu_result,
        add_result_out => add_result,
        new_pc_out => new_pc,
        pc_src_out => pc_src,
        rd_data_out => rd_data,
        rd_addr_out => rd_addr,
        reg_write_out => reg_write
    );

    -- combine signals to pass to make it easier to pass to the compiler
    clk <= signals(65);
    write_en <= signals(64);
    write_inst <= signals(63 downto 32);
    write_addr <= signals(31 downto 0);
```

```
    process begin
        -- 0
        lui(31, 1, signals); -- x31 = 4096 (imm<<12)
        -- 4
        nop(signals);
        -- 8
        jal(1, 8, signals); -- PC = 24 (PC + (2 * imm))
        -- 12
        nop(signals);
        -- 16
        sw(30, 31, 0, signals); -- data_addr[16 + 0] = 4096 (data_addr[rs1 +
imm] = rs2)
        -- 20
        lw(29, 30, 0, signals); -- x29 = data_addr[16 + 0] (data_addr[rs1 +
imm] = rs2)
        -- 24
        srli(30, 31, 8, signals); -- x30 = 4096 >> 8 (x31 >> imm) = 16
        -- 28
        nop(signals);
        -- 32
        nop(signals);
        -- 36
        bne(30, 31, -10, signals); -- if x30 != x31, PC = 16 (PC + (2 * imm))
        run(signals);
    end process;
end behaviour;
```

11.5 Data_Extender_testbench.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;

entity data_extender_testbench is
end data_extender_testbench;

-- define the internal organisation and operation of the data extender
testbench
architecture behaviour of data_extender_testbench is
    -- architecture declarations
    constant time_delta : time := 100 ns;

    signal oper, oper_out   : std_logic_vector(31 downto 0) := (others =>
'0');
    signal funct            : std_logic_vector(2 downto 0)  := (others =>
'0');
begin

    data_extender : entity work.u32_data_extender
    port map (
        funct => funct,
        oper => oper,
        oper_out => oper_out
    );

    process
        procedure test(
            constant inst_type      : in string;
            constant passed_funct   : in std_logic_vector(2 downto 0);
            constant operand        : in std_logic_vector(31 downto 0);
            constant expected       : in std_logic_vector(31 downto 0)
        ) is
            variable res : integer;
        begin
            funct <= passed_funct;
            oper <= operand;

            wait for time_delta;

            assert oper_out = expected
            report "unexpected result: " &
            inst_type & "; " &
            "operand = " & to_hex_string(oper_out) & "; " &
            "expected = " & to_hex_string(expected)
            severity error;
        end procedure test;
```

```
    procedure test_lb(
        constant operand    : in std_logic_vector(31 downto 0);
        constant expected   : in std_logic_vector(31 downto 0)
    ) is
        constant inst_type  : string := "instruction_type = LB";
    begin
        test(inst_type, "000", operand, expected);
    end procedure test_lb;

    procedure test_lh(
        constant operand    : in std_logic_vector(31 downto 0);
        constant expected   : in std_logic_vector(31 downto 0)
    ) is
        constant inst_type  : string := "instruction_type = LH";
    begin
        test(inst_type, "001", operand, expected);
    end procedure test_lh;

    procedure test_lw(
        constant operand    : in std_logic_vector(31 downto 0)
    ) is
        constant inst_type  : string := "instruction_type = LW";
    begin
        test(inst_type, "010", operand, operand);
    end procedure test_lw;

    procedure test_lbu(
        constant operand    : in std_logic_vector(31 downto 0);
        constant expected   : in std_logic_vector(31 downto 0)
    ) is
        constant inst_type  : string := "instruction_type = LBU";
    begin
        test(inst_type, "100", operand, expected);
    end procedure test_lbu;

    procedure test_lhu(
        constant operand    : in std_logic_vector(31 downto 0);
        constant expected   : in std_logic_vector(31 downto 0)
    ) is
        constant inst_type  : string := "instruction_type = LHU";
    begin
        test(inst_type, "101", operand, expected);
    end procedure test_lhu;
begin
    test_lb(x"00000000", x"00000000");
    test_lb(x"55555555", x"00000055");
    test_lb(x"AAAAAAAA", x"FFFFFFAA");
```

```
        test_lb(x"FFFFFFFF", x"FFFFFFFF");

        test_lh(x"00000000", x"00000000");
        test_lh(x"55555555", x"00005555");
        test_lh(x"AAAAAAAA", x"FFFFAAAA");
        test_lh(x"FFFFFFFF", x"FFFFFFFF");

        test_lw(x"00000000");
        test_lw(x"55555555");
        test_lw(x"AAAAAAAA");
        test_lw(x"FFFFFFFF");

        test_lbu(x"00000000", x"00000000");
        test_lbu(x"55555555", x"00000055");
        test_lbu(x"AAAAAAAA", x"000000AA");
        test_lbu(x"FFFFFFFF", x"000000FF");

        test_lhu(x"00000000", x"00000000");
        test_lhu(x"55555555", x"00005555");
        test_lhu(x"AAAAAAAA", x"0000AAAA");
        test_lhu(x"FFFFFFFF", x"0000FFFF");
        wait;
    end process;
end behaviour;
```

11.6 Data_Memory_testbench.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity data_memory_testbench is
end data_memory_testbench;

-- define the internal organisation and operation of the data memory testbench
architecture behaviour of data_memory_testbench is
    -- architecture declarations
    constant clock_delay    : time := 50 ns;

    signal clk, write_en, read_en     : std_logic := '0';
    signal addr, write_data, read_data  : std_logic_vector(31 downto 0) :=
(others => '0');
    signal funct                        : std_logic_vector(1 downto 0) :=
(others => '0');
-- concurrent statements
begin
    -- instantiate alu_controller
    data_memory : entity work.u32_data_memory
    port map (
        clk => clk,
        write_en => write_en,
        read_en => read_en,
        funct => funct,
        addr => addr,
        write_data => write_data,
        read_data => read_data
    );

    process
        procedure test_read(
            constant address, expected  : in std_logic_vector(31 downto 0)
        ) is begin
            -- assign values to circuit inputs
            addr <= address;
            read_en <= '1';
            write_en <= '0';

            -- cycle clock
            wait for clock_delay;
            clk <= '1';
            wait for clock_delay;
            clk <= '0';

            -- log any unexpected outputs
```

```
            assert read_data = expected
            report "Unexpected data: " &
            "data = 0x" & to_hex_string(read_data) & "; " &
            "expected = 0x" & to_hex_string(expected) & "; "
            severity error;
        end procedure test_read;

        procedure test_write(
            constant address, data  : in std_logic_vector(31 downto 0);
            constant size           : in std_logic_vector(1 downto 0)
        ) is begin
            addr <= address;
            read_en <= '0';
            write_en <= '1';
            write_data <= data;
            funct <= size;

            wait for clock_delay;
            clk <= '1';
            wait for clock_delay;
            clk <= '0';
        end procedure test_write;
    begin
        -- test vectors set based on data mem size of 256B
        test_read(x"00000000", x"00000000");
        test_write(x"00000000", x"FFFFFFFF", "10");
        test_read(x"00000000", x"FFFFFFFF");

        test_read(x"00000009", x"00000000");
        test_write(x"00000009", x"FFFFFFFF", "10");
        test_read(x"00000009", x"FFFFFFFF");

        test_read(x"000000FC", x"00000000");
        test_read(x"000000FD", x"00000000");
        test_read(x"000000FE", x"00000000");
        test_read(x"000000FF", x"00000000");
        test_write(x"000000FC", x"FF55AA11", "10");
        test_read(x"000000FC", x"FF55AA11");
        test_read(x"000000FD", x"00FF55AA");
        test_read(x"000000FE", x"0000FF55");
        test_read(x"000000FF", x"000000FF");

        test_write(x"000000FD", x"44444444", "10");
        test_read(x"000000FD", x"00444444");
        test_read(x"000000FC", x"44444411");
        test_write(x"000000FE", x"AAAAAAAA", "10");
        test_read(x"000000FE", x"0000AAAA");
        test_read(x"000000FC", x"AAAA4411");
```

```
        test_write(x"000000FF", x"FFFFFFFF", "10");
        test_read(x"000000FF", x"000000FF");
        test_read(x"000000FC", x"FFAA4411");

        test_write(x"000000FC", x"FFFFFFFF", "01");
        test_read(x"000000FC", x"FFAAFFFF");
        test_write(x"000000FD", x"55555555", "01");
        test_read(x"000000FC", x"FF5555FF");
        test_write(x"000000FE", x"AAAAAAAA", "01");
        test_read(x"000000FC", x"AAAA55FF");
        test_write(x"000000FF", x"11111111", "01");
        test_read(x"000000FC", x"11AA55FF");

        test_write(x"000000FC", x"44444444", "00");
        test_read(x"000000FC", x"11AA5544");
        test_write(x"000000FD", x"FFFFFFFF", "00");
        test_read(x"000000FC", x"11AAFF44");
        test_write(x"000000FE", x"55555555", "00");
        test_read(x"000000FC", x"1155FF44");
        test_write(x"000000FF", x"AAAAAAAA", "00");
        test_read(x"000000FC", x"AA55FF44");
        wait;
    end process;
end behaviour;
```

11.7 Decode_testbench.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity decode_testbench is
end decode_testbench;

-- define the internal organisation and operation of the decode pipeline stage
testbench
architecture behaviour of decode_testbench is
    -- architecture declarations
    constant clock_delay    : time := 50 ns;
    signal current_pc : std_logic_vector(31 downto 0) := (others => '0');

    signal clk, clk_en, reg_write                   : std_logic
:= '0';
    signal rd_addr, rd                              : std_logic_vector(4
downto 0)   := (others => '0');
    signal inst                                     : std_logic_vector(29
downto 0)  := (others => '0');
    signal pc_in, next_pc_in, rd_data, rs1d, rs2d,
           imm, pc_out, next_pc_out                 : std_logic_vector(31
downto 0)  := (others => '0');
    signal control                                  : std_logic_vector(8
downto 0)   := (others => '0');
    signal funct                                    : std_logic_vector(3
downto 0)   := (others => '0');

    function check_control(
        opcode : std_logic_vector(4 downto 0);
        control_signal : std_logic_vector(8 downto 0)
    )
        return std_ulogic is
    begin
        case opcode is
            when "01101" => -- LUI
                if control_signal = "100001000" then
                    return '1';
                else
                    return '0';
                end if;
            when "00101" => -- AUIPC
                if control_signal = "100010000" then
                    return '1';
                else
                    return '0';
                end if;
```

```
            when "11011" => -- JAL
                if control_signal = "100000001" then
                    return '1';
                else
                    return '0';
                end if;
            when "11001" => -- JALR
                if control_signal = "100000101" then
                    return '1';
                else
                    return '0';
                end if;
            when "11000" => -- BRANCH
                if control_signal = "000011010" then
                    return '1';
                else
                    return '0';
                end if;
            when "00000" => -- LOAD
                if control_signal = "110110100" then
                    return '1';
                else
                    return '0';
                end if;
            when "01000" => -- STORE
                if control_signal = "001011100" then
                    return '1';
                else
                    return '0';
                end if;
            when "01100" => -- OP
                if control_signal = "100101100" then
                    return '1';
                else
                    return '0';
                end if;
            when others => -- OP-IMM
                if control_signal = "100100000" then
                    return '1';
                else
                    return '0';
                end if;
        end case;
    end check_control;
-- concurrent statements
begin
    -- instantiate alu_controller
    decode : entity work.u32_decode
```

```
    port map (
        -- inputs
        clk => clk,
        clk_en => clk_en,
        reg_write => reg_write,
        rd_addr => rd_addr,
        inst => inst,
        pc_in => pc_in,
        next_pc_in => next_pc_in,
        rd_data => rd_data,
        -- outputs
        rd => rd,
        rs1d => rs1d,
        rs2d => rs2d,
        pc_out => pc_out,
        next_pc_out => next_pc_out,
        control => control,
        funct => funct,
        imm => imm
    );

    process
        procedure test(
            constant instruction, data  : in std_logic_vector(31 downto 0);
            constant address            : in std_logic_vector(4 downto 0);
            constant regwrite, en_clk   : in std_logic
        ) is

        begin
            pc_in <= next_pc_out;
            next_pc_in <= next_pc_out + x"00000004";
            reg_write <= regwrite;
            rd_data <= data;
            rd_addr <= address;
            inst <=  instruction(31 downto 2);
            clk_en <= en_clk;

            if en_clk = '1' then
                current_pc <= next_pc_out;
            end if;

            wait for clock_delay;
            clk <= '1';
            wait for clock_delay;
            clk <= '0';

            wait for 1 ps;
```

```
            if (en_clk = '1') and (current_pc > x"00000004") then
                assert (pc_out = current_pc and next_pc_out = (current_pc +
x"00000004"))
                report "Unexcpected PC: " &
                "pc_out = 0x" & to_hex_string(pc_out) & "; " &
                "current_pc = 0x" & to_hex_string(current_pc) & "; "
                severity error;

                assert check_control(instruction(6 downto 2), control)
                report "Unexcpected Control: " &
                "pc_out = 0x" & to_hex_string(pc_out) & "; " &
                "current_pc = 0x" & to_hex_string(current_pc) & "; "
                severity error;
            end if;

            report "Tested inst: " &
            "rs1d = 0x" & to_hex_string(rs1d) & "; " &
            "rs2d = 0x" & to_hex_string(rs2d) & "; " &
            "funct = " & to_string(funct) & "; " &
            "imm = 0x" & to_hex_string(imm) & "; " &
            "rd = " & to_string(rd) & "; "
            severity note;
        end procedure test;
    begin
        -- test pipeline registers are disabled unless clk_en is set high
        test(x"FFFFFFFF", x"00000000", "00000", '0', '0');
        -- write into register x1 and test U-Type immediates decode
        test(x"401120B7", x"FFFFFFFF", "00001", '1', '1');
        -- write into register x2
        test(x"401120B3", x"FFFFFFFF", "00010", '1', '1');
        wait for 10 ns;
        wait;
    end process;
end behaviour;
```

11.8 Execute_testbench.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity execute_testbench is
end execute_testbench;

-- define the internal organisation and operation of the execute pipeline
stage testbench
architecture behaviour of execute_testbench is
    -- architecture declarations
    constant clock_delay    : time := 50 ns;

    signal clk, clk_en, pc_src              : std_logic
:= '0';
    signal control                          : std_logic_vector(8 downto 0)
:= (others => '0');
    signal rd, control_out, rd_out          : std_logic_vector(4 downto 0)
:= (others => '0');
    signal funct, funct_out                 : std_logic_vector(3 downto 0)
:= (others => '0');
    signal rs1d, rs2d, imm, pc, next_pc,
            alu_result, add_result,
            add_result_out, addr_const      : std_logic_vector(31 downto 0)
:= (others => '0');
-- concurrent statements
begin
    -- instantiate alu_controller
    execute : entity work.u32_execute
    port map (
        -- inputs
        clk => clk,
        clk_en => clk_en,
        control => control,
        rd => rd,
        funct => funct,
        rs1d => rs1d,
        rs2d => rs2d,
        imm => imm,
        pc => pc,
        next_pc => next_pc,
        -- outputs
        control_out => control_out,
        rd_out => rd_out,
        funct_out => funct_out,
        alu_result => alu_result,
        add_result => add_result,
```

```
        add_result_out => add_result_out,
        addr_const => addr_const,
        pc_src => pc_src
    );

    process
        procedure test(
            constant passed_rs1d, passed_rs2d, passed_imm   : in
std_logic_vector(31 downto 0);
            constant passed_pc_src                           : in std_logic;
            constant passed_funct                           : in
std_logic_vector(3 downto 0);
            constant inst_type                              : in string
        ) is
        begin
            funct <= passed_funct;
            rs1d <= passed_rs1d;
            rs2d <= passed_rs2d;
            imm <= passed_imm;
            rd <= "11111";
            clk_en <= '1';

            wait for clock_delay;
            clk <= '1';
            wait for clock_delay;
            clk <= '0';

            wait for 1 ps;
            assert funct_out = passed_funct
            report "Unexcpected result: " &
            "instruction type = " & inst_type & "; " &
            "funct_out = " & to_string(funct_out) & "; " &
            "expected = 1111; "
            severity error;

            assert rd_out = "11111"
            report "Unexcpected result: " &
            "instruction type = " & inst_type & "; " &
            "rd_out = " & to_string(rd_out) & "; " &
            "expected = 11111; "
            severity error;

            assert pc_src = passed_pc_src
            report "Unexcpected result: " &
            "instruction type = " & inst_type & "; " &
            "pc_src = " & to_string(pc_src) & "; " &
            "expected = " & to_string(passed_pc_src) & "; "
            severity error;
```

```
        end procedure test;

        procedure test_lui(
            constant passed_imm    : in std_logic_vector(31 downto 0)
        ) is
            constant inst_type : string := "LUI";
        begin
            control <= "100001000";
            test(x"FFFFFFFF", x"FFFFFFFF", passed_imm, '0', "1111",
inst_type);
            assert addr_const = passed_imm
            report "Unexcpected result: " &
            "instruction type = " & inst_type & "; " &
            "addr_const = " & to_hex_string(addr_const) & "; " &
            "expected = " & to_hex_string(passed_imm) & "; "
            severity error;
        end procedure test_lui;

        procedure test_auipc(
            constant passed_imm, passed_pc, expected   : in
std_logic_vector(31 downto 0)
        ) is
            constant inst_type : string := "AUIPC";
        begin
            control <= "100010000";
            pc <= passed_pc;
            test(x"FFFFFFFF", x"FFFFFFFF", passed_imm, '0', "1111",
inst_type);
            assert add_result = expected
            report "Unexcpected result: " &
            "instruction type = AUIPC; " &
            "add_result = " & to_hex_string(add_result) & "; " &
            "expected = " & to_hex_string(expected) & "; "
            severity error;
        end procedure test_auipc;

        procedure test_jal(
            constant passed_imm, passed_pc, expected   : in
std_logic_vector(31 downto 0)
        ) is
            constant inst_type : string := "JAL";
        begin
            control <= "100000001";
            pc <= passed_pc;
            next_pc <= passed_pc + x"00000004";
            test(x"FFFFFFFF", x"FFFFFFFF", passed_imm, '1', "1111",
inst_type);
            assert addr_const = (passed_pc + x"00000004")
```

```
            report "Unexcpected result: " &
            "instruction type = " & inst_type & "; " &
            "addr_const = " & to_hex_string(addr_const) & "; " &
            "expected = " & to_hex_string(passed_pc + x"00000004") & "; "
            severity error;
            assert add_result_out = expected
            report "Unexcpected result: " &
            "instruction type = " & inst_type & "; " &
            "add_result_out = " & to_hex_string(add_result_out) & "; " &
            "expected = " & to_hex_string(expected) & "; "
            severity error;
        end procedure test_jal;

        procedure test_jalr(
            constant passed_rs1d, passed_imm, expected  : in
std_logic_vector(31 downto 0)
        ) is
            constant inst_type : string := "JALR";
        begin
            control <= "100000101";
            next_pc <= x"FFFFFFFF";
            test(passed_rs1d, x"FFFFFFFF", passed_imm, '1', "1111",
inst_type);
            assert addr_const = x"FFFFFFFF"
            report "Unexcpected result: " &
            "instruction type = " & inst_type & "; " &
            "addr_const = " & to_hex_string(addr_const) & "; " &
            "expected = 0xFFFFFFFF; "
            severity error;
            assert add_result_out = expected
            report "Unexcpected result: " &
            "instruction type = " & inst_type & "; " &
            "add_result_out = " & to_hex_string(add_result_out) & "; " &
            "expected = " & to_hex_string(expected) & "; "
            severity error;
        end procedure test_jalr;

        procedure test_branch(
            constant passed_rs1d, passed_rs2d, passed_imm, passed_pc, expected
: in std_logic_vector(31 downto 0);
            constant branch_taken  : in std_logic;
            constant passed_funct   : in std_logic_vector(3 downto 0)
        ) is
            constant inst_type : string := "BRANCH";
        begin
            control <= "000011010";
            pc <= passed_pc;
```

```
            test(passed_rs1d, passed_rs2d, passed_imm, branch_taken,
passed_funct, inst_type);
            assert add_result_out = expected
            report "Unexcpected result: " &
            "instruction type = " & inst_type & "; " &
            "funct = " & to_string(passed_funct) & "; " &
            "add_result_out = " & to_hex_string(add_result_out) & "; " &
            "expected = " & to_hex_string(expected) & "; "
            severity error;
        end procedure test_branch;

        procedure test_load(
            constant passed_rs1d, passed_imm, expected  : in
std_logic_vector(31 downto 0)
        ) is
            constant inst_type : string := "LOAD";
        begin
            control <= "110110100";
            test(passed_rs1d, x"FFFFFFFF", passed_imm, '0', "1111",
inst_type);
            assert add_result = expected
            report "Unexcpected result: " &
            "instruction type = " & inst_type & "; " &
            "add_result = " & to_hex_string(add_result) & "; " &
            "expected = " & to_hex_string(expected) & "; "
            severity error;
        end procedure test_load;

        procedure test_store(
            constant passed_rs1d, passed_rs2d, passed_imm, expected  : in
std_logic_vector(31 downto 0);
            constant passed_funct   : in std_logic_vector(3 downto 0)
        ) is
            constant inst_type : string := "STORE";
        begin
            control <= "001011100";
            test(passed_rs1d, passed_rs2d, passed_imm, '0', passed_funct,
inst_type);
            assert add_result = expected
            report "Unexcpected result: " &
            "instruction type = " & inst_type & "; " &
            "funct = " & to_string(passed_funct) & "; " &
            "add_result = " & to_hex_string(add_result) & "; " &
            "expected = " & to_hex_string(expected) & "; "
            severity error;
            assert alu_result = passed_rs2d
            report "Unexcpected result: " &
            "instruction type = " & inst_type & "; " &
```

```
            "funct = " & to_string(passed_funct) & "; " &
            "alu_result = " & to_hex_string(alu_result) & "; " &
            "expected = " & to_hex_string(passed_rs2d) & "; "
            severity error;
        end procedure test_store;

        procedure test_opimm(
            constant passed_rs1d, passed_imm, expected  : in
std_logic_vector(31 downto 0);
            constant passed_funct   : in std_logic_vector(3 downto 0)
        ) is
            constant inst_type : string := "OP-IMM";
        begin
            control <= "100100000";
            test(passed_rs1d, x"FFFFFFFF", passed_imm, '0', passed_funct,
inst_type);
            assert alu_result = expected
            report "Unexcpected result: " &
            "instruction type = " & inst_type & "; " &
            "funct = " & to_string(passed_funct) & "; " &
            "alu_result = " & to_hex_string(alu_result) & "; " &
            "expected = " & to_hex_string(expected) & "; "
            severity error;
        end procedure test_opimm;

        procedure test_op(
            constant passed_rs1d, passed_rs2d, expected  : in
std_logic_vector(31 downto 0);
            constant passed_funct   : in std_logic_vector(3 downto 0)
        ) is
            constant inst_type : string := "OP";
        begin
            control <= "100101100";
            test(passed_rs1d, passed_rs2d, x"FFFFFFFF", '0', passed_funct,
inst_type);
            assert alu_result = expected
            report "Unexcpected result: " &
            "instruction type = " & inst_type & "; " &
            "funct = " & to_string(passed_funct) & "; " &
            "alu_result = " & to_hex_string(alu_result) & "; " &
            "expected = " & to_hex_string(expected) & "; "
            severity error;
        end procedure test_op;
    begin
        test_lui(x"401000B3");

        test_auipc(x"401000B3", x"00000002", x"401000B5");
```

```
        test_jal(x"401000B3", x"00000001", x"401000B4");

        test_jalr(x"401000B3", x"00000001", x"401000B4");

        test_branch(x"401000B3", x"401000B3", x"00000001", x"00000001",
x"00000002", '1', "0000"); -- BEQ
        test_branch(x"401000B3", x"401000B4", x"10000002", x"30000002",
x"40000004", '0', "1000"); -- BEQ
        test_branch(x"401000B3", x"401000B3", x"00000001", x"00000001",
x"00000002", '0', "0001"); -- BNE
        test_branch(x"401000B3", x"401000B4", x"10000002", x"30000002",
x"40000004", '1', "1001"); -- BNE
        test_branch(x"FFFFFFFF", x"00000000", x"00000001", x"00000001",
x"00000002", '1', "0100"); -- BLT
        test_branch(x"00000000", x"00000000", x"10000002", x"30000002",
x"40000004", '0', "1100"); -- BLT
        test_branch(x"FFFFFFFF", x"00000000", x"00000001", x"00000001",
x"00000002", '0', "0101"); -- BGE
        test_branch(x"00000000", x"00000000", x"10000002", x"30000002",
x"40000004", '1', "1101"); -- BGE
        test_branch(x"FFFFFFFF", x"00000000", x"00000001", x"00000001",
x"00000002", '0', "0110"); -- BLTU
        test_branch(x"00000000", x"00000000", x"10000002", x"30000002",
x"40000004", '0', "1110"); -- BLTU
        test_branch(x"FFFFFFFF", x"00000000", x"00000001", x"00000001",
x"00000002", '1', "0111"); -- BGEU
        test_branch(x"00000000", x"00000000", x"10000002", x"30000002",
x"40000004", '1', "1111"); -- BGEU

        test_load(x"55555555", x"AAAAAAAA", x"FFFFFFFF");

        test_store(x"00000000", x"00000000", x"00000000", x"00000000",
"0000"); -- SB
        test_store(x"00000000", x"00000001", x"00000001", x"00000001",
"1000"); -- SB
        test_store(x"00000001", x"10000000", x"00000001", x"00000002",
"0001"); -- SH
        test_store(x"55555555", x"55555555", x"AAAAAAAA", x"FFFFFFFF",
"1001"); -- SH
        test_store(x"AAAAAAAA", x"AAAAAAAA", x"55555555", x"FFFFFFFF",
"0010"); -- SW
        test_store(x"FFFFFFFE", x"FFFFFFFF", x"00000001", x"FFFFFFFF",
"1010"); -- SW

        test_opimm(x"55555555", x"AAAAAAAA", x"FFFFFFFF", "0000"); -- ADDI
        test_opimm(x"55555555", x"AAAAAAAA", x"FFFFFFFF", "1000"); -- ADDI
        test_opimm(x"FFFFFFFF", x"00000000", x"00000001", "0010"); -- SLTI
        test_opimm(x"FFFFFFFF", x"00000000", x"00000001", "1010"); -- SLTI
```

```
        test_opimm(x"FFFFFFFF", x"00000000", x"00000000", "0011"); -- SLTUI
        test_opimm(x"FFFFFFFF", x"00000000", x"00000000", "1011"); -- SLTUI
        test_opimm(x"FFFFFFFF", x"FFFFFFFF", x"00000000", "0100"); -- XORI
        test_opimm(x"FFFFFFFF", x"FFFFFFFF", x"00000000", "1100"); -- XORI
        test_opimm(x"FFFFFFFF", x"00000000", x"FFFFFFFF", "0110"); -- ORI
        test_opimm(x"FFFFFFFF", x"00000000", x"FFFFFFFF", "1110"); -- ORI
        test_opimm(x"FFFFFFFF", x"00000000", x"00000000", "0111"); -- ANDI
        test_opimm(x"FFFFFFFF", x"00000000", x"00000000", "1111"); -- ANDI
        test_opimm(x"55555555", x"00000001", x"AAAAAAAA", "0001"); -- SLLI
        test_opimm(x"AAAAAAAA", x"00000001", x"55555555", "0101"); -- SRLI
        test_opimm(x"AAAAAAAA", x"00000001", x"D5555555", "1101"); -- SRAI

        test_op(x"55555555", x"AAAAAAAA", x"FFFFFFFF", "0000"); -- ADD
        test_op(x"FFFFFFFF", x"55555555", x"AAAAAAAA", "1000"); -- SUB
        test_op(x"55555555", x"00000001", x"AAAAAAAA", "0001"); -- SLL
        test_op(x"FFFFFFFF", x"00000000", x"00000001", "0010"); -- SLT
        test_op(x"FFFFFFFF", x"00000000", x"00000000", "0011"); -- SLTU
        test_op(x"FFFFFFFF", x"FFFFFFFF", x"00000000", "0100"); -- XOR
        test_op(x"AAAAAAAA", x"00000001", x"55555555", "0101"); -- SRL
        test_op(x"AAAAAAAA", x"00000001", x"D5555555", "1101"); -- SRA
        test_op(x"FFFFFFFF", x"00000000", x"FFFFFFFF", "0110"); -- OR
        test_op(x"FFFFFFFF", x"00000000", x"00000000", "0111"); -- AND
        wait for 10 ns;
        wait;
    end process;
end behaviour;
```

11.9 getto_compiler.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;

package getto_compiler is
    constant clock_delay    : time := 10 ns;

    subtype reg_addr is natural range 0 to 31;
    subtype i_imm is integer range -2048 to 2047;
    subtype load_store_imm is natural range 0 to 4096;
    subtype u_imm is integer range -524288 to 524287;
    subtype j_imm is integer range -524288 to 524287;
    subtype b_imm is integer range  -2048 to 2047;
    subtype shamt_imm is natural range 0 to 31;

    procedure run(
        signal signals  : out std_logic_vector(65 downto 0)
    );

    procedure nop(
        signal signals  : out std_logic_vector(65 downto 0)
    );

    procedure lui(
        constant rd     : in reg_addr;
        constant imm    : in u_imm;

        signal signals  : out std_logic_vector(65 downto 0)
    );

    procedure auipc(
        constant rd     : in reg_addr;
        constant imm    : in u_imm;

        signal signals  : out std_logic_vector(65 downto 0)
    );

    procedure jal(
        constant rd     : in reg_addr;
        constant imm    : in j_imm;

        signal signals  : out std_logic_vector(65 downto 0)
    );

    procedure jalr(
        constant rd     : in reg_addr;
```

```
        constant rs1    : in reg_addr;
        constant imm    : in i_imm;

        signal signals  : out std_logic_vector(65 downto 0)
    );

    procedure beq(
        constant rs1    : in reg_addr;
        constant rs2    : in reg_addr;
        constant imm : in b_imm;

        signal signals  : out std_logic_vector(65 downto 0)
    );

    procedure bne(
        constant rs1    : in reg_addr;
        constant rs2    : in reg_addr;
        constant imm : in b_imm;

        signal signals  : out std_logic_vector(65 downto 0)
    );

    procedure blt(
        constant rs1    : in reg_addr;
        constant rs2    : in reg_addr;
        constant imm : in b_imm;

        signal signals  : out std_logic_vector(65 downto 0)
    );

    procedure bge(
        constant rs1    : in reg_addr;
        constant rs2    : in reg_addr;
        constant imm : in b_imm;

        signal signals  : out std_logic_vector(65 downto 0)
    );

    procedure bltu(
        constant rs1    : in reg_addr;
        constant rs2    : in reg_addr;
        constant imm : in b_imm;

        signal signals  : out std_logic_vector(65 downto 0)
    );

    procedure bgeu(
        constant rs1    : in reg_addr;
```

```
        constant rs2    : in reg_addr;
        constant imm : in b_imm;

        signal signals  : out std_logic_vector(65 downto 0)
    );

    procedure lb(
        constant rd     : in reg_addr;
        constant rs1    : in reg_addr;
        constant imm    : in load_store_imm;

        signal signals  : out std_logic_vector(65 downto 0)
    );

    procedure lh(
        constant rd     : in reg_addr;
        constant rs1    : in reg_addr;
        constant imm    : in load_store_imm;

        signal signals  : out std_logic_vector(65 downto 0)
    );

    procedure lw(
        constant rd     : in reg_addr;
        constant rs1    : in reg_addr;
        constant imm    : in load_store_imm;

        signal signals  : out std_logic_vector(65 downto 0)
    );

    procedure lbu(
        constant rd     : in reg_addr;
        constant rs1    : in reg_addr;
        constant imm    : in load_store_imm;

        signal signals  : out std_logic_vector(65 downto 0)
    );

    procedure lhu(
        constant rd     : in reg_addr;
        constant rs1    : in reg_addr;
        constant imm    : in load_store_imm;

        signal signals  : out std_logic_vector(65 downto 0)
    );

    procedure sb(
        constant rs1    : in reg_addr;
```

```
        constant rs2    : in reg_addr;
        constant imm    : in load_store_imm;

        signal signals  : out std_logic_vector(65 downto 0)
    );

    procedure sh(
        constant rs1    : in reg_addr;
        constant rs2    : in reg_addr;
        constant imm    : in load_store_imm;

        signal signals  : out std_logic_vector(65 downto 0)
    );

    procedure sw(
        constant rs1    : in reg_addr;
        constant rs2    : in reg_addr;
        constant imm    : in load_store_imm;

        signal signals  : out std_logic_vector(65 downto 0)
    );

    procedure addi(
        constant rd     : in reg_addr;
        constant rs1    : in reg_addr;
        constant imm    : in i_imm;

        signal signals  : out std_logic_vector(65 downto 0)
    );

    procedure slti(
        constant rd     : in reg_addr;
        constant rs1    : in reg_addr;
        constant imm    : in i_imm;

        signal signals  : out std_logic_vector(65 downto 0)
    );

    procedure sltiu(
        constant rd     : in reg_addr;
        constant rs1    : in reg_addr;
        constant imm    : in i_imm;

        signal signals  : out std_logic_vector(65 downto 0)
    );

    procedure xori(
        constant rd     : in reg_addr;
```

```
        constant rs1    : in reg_addr;
        constant imm    : in i_imm;

        signal signals  : out std_logic_vector(65 downto 0)
    );

    procedure ori(
        constant rd     : in reg_addr;
        constant rs1    : in reg_addr;
        constant imm    : in i_imm;

        signal signals  : out std_logic_vector(65 downto 0)
    );

    procedure andi(
        constant rd     : in reg_addr;
        constant rs1    : in reg_addr;
        constant imm    : in i_imm;

        signal signals  : out std_logic_vector(65 downto 0)
    );

    procedure slli(
        constant rd     : in reg_addr;
        constant rs1    : in reg_addr;
        constant shamt  : in shamt_imm;

        signal signals  : out std_logic_vector(65 downto 0)
    );

    procedure srli(
        constant rd     : in reg_addr;
        constant rs1    : in reg_addr;
        constant shamt  : in shamt_imm;

        signal signals  : out std_logic_vector(65 downto 0)
    );

    procedure srai(
        constant rd     : in reg_addr;
        constant rs1    : in reg_addr;
        constant shamt  : in shamt_imm;

        signal signals  : out std_logic_vector(65 downto 0)
    );

    procedure op_add(
        constant rd     : in reg_addr;
```

```
        constant rs1    : in reg_addr;
        constant rs2    : in reg_addr;

        signal signals  : out std_logic_vector(65 downto 0)
    );

    procedure op_sub(
        constant rd     : in reg_addr;
        constant rs1    : in reg_addr;
        constant rs2    : in reg_addr;

        signal signals  : out std_logic_vector(65 downto 0)
    );

    procedure op_sll(
        constant rd     : in reg_addr;
        constant rs1    : in reg_addr;
        constant rs2    : in reg_addr;

        signal signals  : out std_logic_vector(65 downto 0)
    );

    procedure op_slt(
        constant rd     : in reg_addr;
        constant rs1    : in reg_addr;
        constant rs2    : in reg_addr;

        signal signals  : out std_logic_vector(65 downto 0)
    );

    procedure op_sltu(
        constant rd     : in reg_addr;
        constant rs1    : in reg_addr;
        constant rs2    : in reg_addr;

        signal signals  : out std_logic_vector(65 downto 0)
    );

    procedure op_xor(
        constant rd     : in reg_addr;
        constant rs1    : in reg_addr;
        constant rs2    : in reg_addr;

        signal signals  : out std_logic_vector(65 downto 0)
    );

    procedure op_srl(
        constant rd     : in reg_addr;
```

```
        constant rs1    : in reg_addr;
        constant rs2    : in reg_addr;

        signal signals  : out std_logic_vector(65 downto 0)
    );

    procedure op_sra(
        constant rd     : in reg_addr;
        constant rs1    : in reg_addr;
        constant rs2    : in reg_addr;

        signal signals  : out std_logic_vector(65 downto 0)
    );

    procedure op_or(
        constant rd     : in reg_addr;
        constant rs1    : in reg_addr;
        constant rs2    : in reg_addr;

        signal signals  : out std_logic_vector(65 downto 0)
    );

    procedure op_and(
        constant rd     : in reg_addr;
        constant rs1    : in reg_addr;
        constant rs2    : in reg_addr;

        signal signals  : out std_logic_vector(65 downto 0)
    );
end package getto_compiler;

package body getto_compiler is
    -- procedure only used with the package iteself
    procedure load_inst(
        constant inst     : in std_logic_vector(31 downto 0);

        signal signals  : out std_logic_vector(65 downto 0)
    ) is begin
        signals(64) <= '1'; -- set instruction memory write enable high
        signals(63 downto 32) <= inst; -- write instruction into instruction
memory

        -- cycle clock
        wait for clock_delay;
        signals(65) <= '1';
        wait for clock_delay;
        signals(65) <= '0';
```

```
        -- increment write address for instruction memory
        signals(31 downto 0) <= signals(31 downto 0) + x"00000004";
    end procedure load_inst;

    procedure run(
        signal signals  : out std_logic_vector(65 downto 0)
    ) is begin
        -- stop writing instruction into isntruction memory
        signals(64 downto 0) <= (others => '0');

        -- cycle clock
        for i in 0 to 69 loop
            wait for clock_delay;
            signals(65) <= '1';
            wait for clock_delay;
            signals(65) <= '0';
        end loop;
        wait;
    end procedure run;

    -- pseudo instructions
    procedure nop(
        signal signals  : out std_logic_vector(65 downto 0)
    ) is begin
        addi(0, 0, 0, signals);
    end procedure nop;

    -- instructions
    procedure lui(
        constant rd     : in reg_addr;
        constant imm    : in u_imm;

        signal signals  : out std_logic_vector(65 downto 0)
    ) is
        constant opcode     : std_logic_vector(6 downto 0)  := "0110111";
        variable inst       : std_logic_vector(31 downto 0);
    begin
        inst := std_logic_vector(to_signed(imm, 20)) &
std_logic_vector(to_unsigned(rd, 5)) & opcode;
        load_inst(inst, signals);
    end procedure lui;

    procedure auipc(
        constant rd     : in reg_addr;
        constant imm    : in u_imm;

        signal signals  : out std_logic_vector(65 downto 0)
    ) is
```

```
        constant opcode     : std_logic_vector(6 downto 0)  := "0010111";
        variable inst       : std_logic_vector(31 downto 0);
    begin
        inst := std_logic_vector(to_signed(imm, 20)) &
std_logic_vector(to_unsigned(rd, 5)) & opcode;
        load_inst(inst, signals);
    end procedure auipc;

    procedure jal(
        constant rd     : in reg_addr;
        constant imm    : in j_imm;

        signal signals  : out std_logic_vector(65 downto 0)
    ) is
        constant opcode     : std_logic_vector(6 downto 0)  := "1101111";
        variable inst       : std_logic_vector(31 downto 0);
        variable tmp_imm    : std_logic_vector(20 downto 1);
    begin
        tmp_imm := std_logic_vector(to_signed(imm, 20));
        inst := tmp_imm(20) & tmp_imm(10 downto 1) & tmp_imm(11) & tmp_imm(19
downto 12) & std_logic_vector(to_unsigned(rd, 5)) & opcode;
        load_inst(inst, signals);
    end procedure jal;

    procedure jalr(
        constant rd     : in reg_addr;
        constant rs1    : in reg_addr;
        constant imm    : in i_imm;

        signal signals  : out std_logic_vector(65 downto 0)
    ) is
        constant funct3     : std_logic_vector(14 downto 12)    := "000";
        constant opcode     : std_logic_vector(6 downto 0)      := "1100111";
        variable inst       : std_logic_vector(31 downto 0);
    begin
        inst := std_logic_vector(to_signed(imm, 12)) &
std_logic_vector(to_unsigned(rs1, 5)) & funct3 &
std_logic_vector(to_unsigned(rd, 5)) & opcode;
        load_inst(inst, signals);
    end procedure jalr;

    procedure beq(
        constant rs1    : in reg_addr;
        constant rs2    : in reg_addr;
        constant imm : in b_imm;

        signal signals  : out std_logic_vector(65 downto 0)
    ) is
```

```
        constant funct3     : std_logic_vector(14 downto 12)    := "000";
        constant opcode     : std_logic_vector(6 downto 0)      := "1100011";
        variable inst       : std_logic_vector(31 downto 0);
        variable offset    : std_logic_vector(12 downto 1);
    begin
        offset := std_logic_vector(to_signed(imm, 12));
        inst := offset(12) & offset(10 downto 5) &
std_logic_vector(to_unsigned(rs2, 5)) & std_logic_vector(to_unsigned(rs1, 5))
& funct3 & offset(4 downto 1) & offset(11) & opcode;
        load_inst(inst, signals);
    end procedure beq;

    procedure bne(
        constant rs1    : in reg_addr;
        constant rs2    : in reg_addr;
        constant imm : in b_imm;

        signal signals  : out std_logic_vector(65 downto 0)
    ) is
        constant funct3     : std_logic_vector(14 downto 12)    := "001";
        constant opcode     : std_logic_vector(6 downto 0)      := "1100011";
        variable inst       : std_logic_vector(31 downto 0);
        variable offset    : std_logic_vector(12 downto 1);
    begin
        offset := std_logic_vector(to_signed(imm, 12));
        inst := offset(12) & offset(10 downto 5) &
std_logic_vector(to_unsigned(rs2, 5)) & std_logic_vector(to_unsigned(rs1, 5))
& funct3 & offset(4 downto 1) & offset(11) & opcode;
        load_inst(inst, signals);
    end procedure bne;

    procedure blt(
        constant rs1    : in reg_addr;
        constant rs2    : in reg_addr;
        constant imm : in b_imm;

        signal signals  : out std_logic_vector(65 downto 0)
    ) is
        constant funct3     : std_logic_vector(14 downto 12)    := "100";
        constant opcode     : std_logic_vector(6 downto 0)      := "1100011";
        variable inst       : std_logic_vector(31 downto 0);
        variable offset    : std_logic_vector(12 downto 1);
    begin
        offset := std_logic_vector(to_signed(imm, 12));
        inst := offset(12) & offset(10 downto 5) &
std_logic_vector(to_unsigned(rs2, 5)) & std_logic_vector(to_unsigned(rs1, 5))
& funct3 & offset(4 downto 1) & offset(11) & opcode;
        load_inst(inst, signals);
```

```
    end procedure blt;

    procedure bge(
        constant rs1    : in reg_addr;
        constant rs2    : in reg_addr;
        constant imm : in b_imm;

        signal signals  : out std_logic_vector(65 downto 0)
    ) is
        constant funct3     : std_logic_vector(14 downto 12)    := "101";
        constant opcode     : std_logic_vector(6 downto 0)      := "1100011";
        variable inst       : std_logic_vector(31 downto 0);
        variable offset    : std_logic_vector(12 downto 1);
    begin
        offset := std_logic_vector(to_signed(imm, 12));
        inst := offset(12) & offset(10 downto 5) &
std_logic_vector(to_unsigned(rs2, 5)) & std_logic_vector(to_unsigned(rs1, 5))
& funct3 & offset(4 downto 1) & offset(11) & opcode;
        load_inst(inst, signals);
    end procedure bge;

    procedure bltu(
        constant rs1    : in reg_addr;
        constant rs2    : in reg_addr;
        constant imm : in b_imm;

        signal signals  : out std_logic_vector(65 downto 0)
    ) is
        constant funct3     : std_logic_vector(14 downto 12)    := "110";
        constant opcode     : std_logic_vector(6 downto 0)      := "1100011";
        variable inst       : std_logic_vector(31 downto 0);
        variable offset    : std_logic_vector(12 downto 1);
    begin
        offset := std_logic_vector(to_signed(imm, 12));
        inst := offset(12) & offset(10 downto 5) &
std_logic_vector(to_unsigned(rs2, 5)) & std_logic_vector(to_unsigned(rs1, 5))
& funct3 & offset(4 downto 1) & offset(11) & opcode;
        load_inst(inst, signals);
    end procedure bltu;

    procedure bgeu(
        constant rs1    : in reg_addr;
        constant rs2    : in reg_addr;
        constant imm : in b_imm;

        signal signals  : out std_logic_vector(65 downto 0)
    ) is
        constant funct3     : std_logic_vector(14 downto 12)    := "111";
```

```
        constant opcode     : std_logic_vector(6 downto 0)      := "1100011";
        variable inst       : std_logic_vector(31 downto 0);
        variable offset    : std_logic_vector(12 downto 1);
    begin
        offset := std_logic_vector(to_signed(imm, 12));
        inst := offset(12) & offset(10 downto 5) &
std_logic_vector(to_unsigned(rs2, 5)) & std_logic_vector(to_unsigned(rs1, 5))
& funct3 & offset(4 downto 1) & offset(11) & opcode;
        load_inst(inst, signals);
    end procedure bgeu;

    procedure lb(
        constant rd     : in reg_addr;
        constant rs1    : in reg_addr;
        constant imm    : in load_store_imm;

        signal signals  : out std_logic_vector(65 downto 0)
    ) is
        constant funct3     : std_logic_vector(14 downto 12)    := "000";
        constant opcode     : std_logic_vector(6 downto 0)      := "0000011";
        variable inst       : std_logic_vector(31 downto 0);
    begin
        inst := std_logic_vector(to_unsigned(imm, 12)) &
std_logic_vector(to_unsigned(rs1, 5)) & funct3 &
std_logic_vector(to_unsigned(rd, 5)) & opcode;
        load_inst(inst, signals);
    end procedure lb;

    procedure lh(
        constant rd     : in reg_addr;
        constant rs1    : in reg_addr;
        constant imm    : in load_store_imm;

        signal signals  : out std_logic_vector(65 downto 0)
    ) is
        constant funct3     : std_logic_vector(14 downto 12)    := "001";
        constant opcode     : std_logic_vector(6 downto 0)      := "0000011";
        variable inst       : std_logic_vector(31 downto 0);
    begin
        inst := std_logic_vector(to_unsigned(imm, 12)) &
std_logic_vector(to_unsigned(rs1, 5)) & funct3 &
std_logic_vector(to_unsigned(rd, 5)) & opcode;
        load_inst(inst, signals);
    end procedure lh;

    procedure lw(
        constant rd     : in reg_addr;
        constant rs1    : in reg_addr;
```

```
        constant imm    : in load_store_imm;

        signal signals  : out std_logic_vector(65 downto 0)
    ) is
        constant funct3     : std_logic_vector(14 downto 12)    := "010";
        constant opcode     : std_logic_vector(6 downto 0)      := "0000011";
        variable inst       : std_logic_vector(31 downto 0);
    begin
        inst := std_logic_vector(to_unsigned(imm, 12)) &
std_logic_vector(to_unsigned(rs1, 5)) & funct3 &
std_logic_vector(to_unsigned(rd, 5)) & opcode;
        load_inst(inst, signals);
    end procedure lw;

    procedure lbu(
        constant rd     : in reg_addr;
        constant rs1    : in reg_addr;
        constant imm    : in load_store_imm;

        signal signals  : out std_logic_vector(65 downto 0)
    ) is
        constant funct3     : std_logic_vector(14 downto 12)    := "100";
        constant opcode     : std_logic_vector(6 downto 0)      := "0000011";
        variable inst       : std_logic_vector(31 downto 0);
    begin
        inst := std_logic_vector(to_unsigned(imm, 12)) &
std_logic_vector(to_unsigned(rs1, 5)) & funct3 &
std_logic_vector(to_unsigned(rd, 5)) & opcode;
        load_inst(inst, signals);
    end procedure lbu;

    procedure lhu(
        constant rd     : in reg_addr;
        constant rs1    : in reg_addr;
        constant imm    : in load_store_imm;

        signal signals  : out std_logic_vector(65 downto 0)
    ) is
        constant funct3     : std_logic_vector(14 downto 12)    := "101";
        constant opcode     : std_logic_vector(6 downto 0)      := "0000011";
        variable inst       : std_logic_vector(31 downto 0);
    begin
        inst := std_logic_vector(to_unsigned(imm, 12)) &
std_logic_vector(to_unsigned(rs1, 5)) & funct3 &
std_logic_vector(to_unsigned(rd, 5)) & opcode;
        load_inst(inst, signals);
    end procedure lhu;
```

```
    procedure sb(
        constant rs1    : in reg_addr;
        constant rs2    : in reg_addr;
        constant imm    : in load_store_imm;

        signal signals  : out std_logic_vector(65 downto 0)
    ) is
        constant funct3     : std_logic_vector(14 downto 12)    := "000";
        constant opcode     : std_logic_vector(6 downto 0)      := "0100011";
        variable inst       : std_logic_vector(31 downto 0);
        variable tmp_imm    : std_logic_vector(11 downto 0);
    begin
        tmp_imm := std_logic_vector(to_unsigned(imm, 12));
        inst := tmp_imm(11 downto 5) & std_logic_vector(to_unsigned(rs2, 5)) &
std_logic_vector(to_unsigned(rs1, 5)) & funct3 & tmp_imm(4 downto 0) & opcode;
        load_inst(inst, signals);
    end procedure sb;

    procedure sh(
        constant rs1    : in reg_addr;
        constant rs2    : in reg_addr;
        constant imm    : in load_store_imm;

        signal signals  : out std_logic_vector(65 downto 0)
    ) is
        constant funct3     : std_logic_vector(14 downto 12)    := "001";
        constant opcode     : std_logic_vector(6 downto 0)      := "0100011";
        variable inst       : std_logic_vector(31 downto 0);
        variable tmp_imm    : std_logic_vector(11 downto 0);
    begin
        tmp_imm := std_logic_vector(to_unsigned(imm, 12));
        inst := tmp_imm(11 downto 5) & std_logic_vector(to_unsigned(rs2, 5)) &
std_logic_vector(to_unsigned(rs1, 5)) & funct3 & tmp_imm(4 downto 0) & opcode;
        load_inst(inst, signals);
    end procedure sh;

    procedure sw(
        constant rs1    : in reg_addr;
        constant rs2    : in reg_addr;
        constant imm    : in load_store_imm;

        signal signals  : out std_logic_vector(65 downto 0)
    ) is
        constant funct3     : std_logic_vector(14 downto 12)    := "010";
        constant opcode     : std_logic_vector(6 downto 0)      := "0100011";
        variable inst       : std_logic_vector(31 downto 0);
        variable tmp_imm    : std_logic_vector(11 downto 0);
    begin
```

```
        tmp_imm := std_logic_vector(to_unsigned(imm, 12));
        inst := tmp_imm(11 downto 5) & std_logic_vector(to_unsigned(rs2, 5)) &
std_logic_vector(to_unsigned(rs1, 5)) & funct3 & tmp_imm(4 downto 0) & opcode;
        load_inst(inst, signals);
    end procedure sw;

    procedure addi(
        constant rd     : in reg_addr;
        constant rs1    : in reg_addr;
        constant imm    : in i_imm;

        signal signals  : out std_logic_vector(65 downto 0)
    ) is
        constant funct3     : std_logic_vector(14 downto 12)    := "000";
        constant opcode     : std_logic_vector(6 downto 0)      := "0010011";
        variable inst       : std_logic_vector(31 downto 0);
    begin
        inst := std_logic_vector(to_signed(imm, 12)) &
std_logic_vector(to_unsigned(rs1, 5)) & funct3 &
std_logic_vector(to_unsigned(rd, 5)) & opcode;
        load_inst(inst, signals);
    end procedure addi;

    procedure slti(
        constant rd     : in reg_addr;
        constant rs1    : in reg_addr;
        constant imm    : in i_imm;

        signal signals  : out std_logic_vector(65 downto 0)
    ) is
        constant funct3     : std_logic_vector(14 downto 12)    := "010";
        constant opcode     : std_logic_vector(6 downto 0)      := "0010011";
        variable inst       : std_logic_vector(31 downto 0);
    begin
        inst := std_logic_vector(to_signed(imm, 12)) &
std_logic_vector(to_unsigned(rs1, 5)) & funct3 &
std_logic_vector(to_unsigned(rd, 5)) & opcode;
        load_inst(inst, signals);
    end procedure slti;

    procedure sltiu(
        constant rd     : in reg_addr;
        constant rs1    : in reg_addr;
        constant imm    : in i_imm;

        signal signals  : out std_logic_vector(65 downto 0)
    ) is
        constant funct3     : std_logic_vector(14 downto 12)    := "011";
```

```
        constant opcode     : std_logic_vector(6 downto 0)      := "0010011";
        variable inst       : std_logic_vector(31 downto 0);
    begin
        inst := std_logic_vector(to_signed(imm, 12)) &
std_logic_vector(to_unsigned(rs1, 5)) & funct3 &
std_logic_vector(to_unsigned(rd, 5)) & opcode;
        load_inst(inst, signals);
    end procedure sltiu;

    procedure xori(
        constant rd     : in reg_addr;
        constant rs1    : in reg_addr;
        constant imm    : in i_imm;

        signal signals  : out std_logic_vector(65 downto 0)
    ) is
        constant funct3     : std_logic_vector(14 downto 12)    := "100";
        constant opcode     : std_logic_vector(6 downto 0)      := "0010011";
        variable inst       : std_logic_vector(31 downto 0);
    begin
        inst := std_logic_vector(to_signed(imm, 12)) &
std_logic_vector(to_unsigned(rs1, 5)) & funct3 &
std_logic_vector(to_unsigned(rd, 5)) & opcode;
        load_inst(inst, signals);
    end procedure xori;

    procedure ori(
        constant rd     : in reg_addr;
        constant rs1    : in reg_addr;
        constant imm    : in i_imm;

        signal signals  : out std_logic_vector(65 downto 0)
    ) is
        constant funct3     : std_logic_vector(14 downto 12)    := "110";
        constant opcode     : std_logic_vector(6 downto 0)      := "0010011";
        variable inst       : std_logic_vector(31 downto 0);
    begin
        inst := std_logic_vector(to_signed(imm, 12)) &
std_logic_vector(to_unsigned(rs1, 5)) & funct3 &
std_logic_vector(to_unsigned(rd, 5)) & opcode;
        load_inst(inst, signals);
    end procedure ori;

    procedure andi(
        constant rd     : in reg_addr;
        constant rs1    : in reg_addr;
        constant imm    : in i_imm;
```

```
        signal signals  : out std_logic_vector(65 downto 0)
    ) is
        constant funct3     : std_logic_vector(14 downto 12)    := "111";
        constant opcode     : std_logic_vector(6 downto 0)      := "0010011";
        variable inst       : std_logic_vector(31 downto 0);
    begin
        inst := std_logic_vector(to_signed(imm, 12)) &
std_logic_vector(to_unsigned(rs1, 5)) & funct3 &
std_logic_vector(to_unsigned(rd, 5)) & opcode;
        load_inst(inst, signals);
    end procedure andi;

    procedure slli(
        constant rd     : in reg_addr;
        constant rs1    : in reg_addr;
        constant shamt  : in shamt_imm;

        signal signals  : out std_logic_vector(65 downto 0)
    ) is
        constant funct7     : std_logic_vector(31 downto 25)    := "0000000";
        constant funct3     : std_logic_vector(14 downto 12)    := "001";
        constant opcode     : std_logic_vector(6 downto 0)      := "0010011";
        variable inst       : std_logic_vector(31 downto 0);
    begin
        inst := funct7 & std_logic_vector(to_unsigned(shamt, 5)) &
std_logic_vector(to_unsigned(rs1, 5)) & funct3 &
std_logic_vector(to_unsigned(rd, 5)) & opcode;
        load_inst(inst, signals);
    end procedure slli;

    procedure srli(
        constant rd     : in reg_addr;
        constant rs1    : in reg_addr;
        constant shamt  : in shamt_imm;

        signal signals  : out std_logic_vector(65 downto 0)
    ) is
        constant funct7     : std_logic_vector(31 downto 25)    := "0000000";
        constant funct3     : std_logic_vector(14 downto 12)    := "101";
        constant opcode     : std_logic_vector(6 downto 0)      := "0010011";
        variable inst       : std_logic_vector(31 downto 0);
    begin
        inst := funct7 & std_logic_vector(to_unsigned(shamt, 5)) &
std_logic_vector(to_unsigned(rs1, 5)) & funct3 &
std_logic_vector(to_unsigned(rd, 5)) & opcode;
        load_inst(inst, signals);
    end procedure srli;
```

```
    procedure srai(
        constant rd     : in reg_addr;
        constant rs1    : in reg_addr;
        constant shamt  : in shamt_imm;

        signal signals  : out std_logic_vector(65 downto 0)
    ) is
        constant funct7     : std_logic_vector(31 downto 25)    := "0100000";
        constant funct3     : std_logic_vector(14 downto 12)    := "101";
        constant opcode     : std_logic_vector(6 downto 0)      := "0010011";
        variable inst       : std_logic_vector(31 downto 0);
    begin
        inst := funct7 & std_logic_vector(to_unsigned(shamt, 5)) &
std_logic_vector(to_unsigned(rs1, 5)) & funct3 &
std_logic_vector(to_unsigned(rd, 5)) & opcode;
        load_inst(inst, signals);
    end procedure srai;

    procedure op_add(
        constant rd     : in reg_addr;
        constant rs1    : in reg_addr;
        constant rs2    : in reg_addr;

        signal signals  : out std_logic_vector(65 downto 0)
    ) is
        constant funct7     : std_logic_vector(31 downto 25)    := "0000000";
        constant funct3     : std_logic_vector(14 downto 12)    := "000";
        constant opcode     : std_logic_vector(6 downto 0)      := "0110011";
        variable inst       : std_logic_vector(31 downto 0);
    begin
        inst := funct7 & std_logic_vector(to_unsigned(rs2, 5)) &
std_logic_vector(to_unsigned(rs1, 5)) & funct3 &
std_logic_vector(to_unsigned(rd, 5)) & opcode;
        load_inst(inst, signals);
    end procedure op_add;

    procedure op_sub(
        constant rd     : in reg_addr;
        constant rs1    : in reg_addr;
        constant rs2    : in reg_addr;

        signal signals  : out std_logic_vector(65 downto 0)
    ) is
        constant funct7     : std_logic_vector(31 downto 25)    := "0100000";
        constant funct3     : std_logic_vector(14 downto 12)    := "000";
        constant opcode     : std_logic_vector(6 downto 0)      := "0110011";
        variable inst       : std_logic_vector(31 downto 0);
    begin
```

```
        inst := funct7 & std_logic_vector(to_unsigned(rs2, 5)) &
std_logic_vector(to_unsigned(rs1, 5)) & funct3 &
std_logic_vector(to_unsigned(rd, 5)) & opcode;
        load_inst(inst, signals);
    end procedure op_sub;

    procedure op_sll(
        constant rd     : in reg_addr;
        constant rs1    : in reg_addr;
        constant rs2    : in reg_addr;

        signal signals  : out std_logic_vector(65 downto 0)
    ) is
        constant funct7     : std_logic_vector(31 downto 25)    := "0000000";
        constant funct3     : std_logic_vector(14 downto 12)    := "001";
        constant opcode     : std_logic_vector(6 downto 0)      := "0110011";
        variable inst       : std_logic_vector(31 downto 0);
    begin
        inst := funct7 & std_logic_vector(to_unsigned(rs2, 5)) &
std_logic_vector(to_unsigned(rs1, 5)) & funct3 &
std_logic_vector(to_unsigned(rd, 5)) & opcode;
        load_inst(inst, signals);
    end procedure op_sll;

    procedure op_slt(
        constant rd     : in reg_addr;
        constant rs1    : in reg_addr;
        constant rs2    : in reg_addr;

        signal signals  : out std_logic_vector(65 downto 0)
    ) is
        constant funct7     : std_logic_vector(31 downto 25)    := "0000000";
        constant funct3     : std_logic_vector(14 downto 12)    := "010";
        constant opcode     : std_logic_vector(6 downto 0)      := "0110011";
        variable inst       : std_logic_vector(31 downto 0);
    begin
        inst := funct7 & std_logic_vector(to_unsigned(rs2, 5)) &
std_logic_vector(to_unsigned(rs1, 5)) & funct3 &
std_logic_vector(to_unsigned(rd, 5)) & opcode;
        load_inst(inst, signals);
    end procedure op_slt;

    procedure op_sltu(
        constant rd     : in reg_addr;
        constant rs1    : in reg_addr;
        constant rs2    : in reg_addr;

        signal signals  : out std_logic_vector(65 downto 0)
```

```
    ) is
        constant funct7     : std_logic_vector(31 downto 25)    := "0000000";
        constant funct3     : std_logic_vector(14 downto 12)    := "011";
        constant opcode     : std_logic_vector(6 downto 0)      := "0110011";
        variable inst       : std_logic_vector(31 downto 0);
    begin
        inst := funct7 & std_logic_vector(to_unsigned(rs2, 5)) &
std_logic_vector(to_unsigned(rs1, 5)) & funct3 &
std_logic_vector(to_unsigned(rd, 5)) & opcode;
        load_inst(inst, signals);
    end procedure op_sltu;

    procedure op_xor(
        constant rd     : in reg_addr;
        constant rs1    : in reg_addr;
        constant rs2    : in reg_addr;

        signal signals  : out std_logic_vector(65 downto 0)
    ) is
        constant funct7     : std_logic_vector(31 downto 25)    := "0000000";
        constant funct3     : std_logic_vector(14 downto 12)    := "100";
        constant opcode     : std_logic_vector(6 downto 0)      := "0110011";
        variable inst       : std_logic_vector(31 downto 0);
    begin
        inst := funct7 & std_logic_vector(to_unsigned(rs2, 5)) &
std_logic_vector(to_unsigned(rs1, 5)) & funct3 &
std_logic_vector(to_unsigned(rd, 5)) & opcode;
        load_inst(inst, signals);
    end procedure op_xor;

    procedure op_srl(
        constant rd     : in reg_addr;
        constant rs1    : in reg_addr;
        constant rs2    : in reg_addr;

        signal signals  : out std_logic_vector(65 downto 0)
    ) is
        constant funct7     : std_logic_vector(31 downto 25)    := "0000000";
        constant funct3     : std_logic_vector(14 downto 12)    := "101";
        constant opcode     : std_logic_vector(6 downto 0)      := "0110011";
        variable inst       : std_logic_vector(31 downto 0);
    begin
        inst := funct7 & std_logic_vector(to_unsigned(rs2, 5)) &
std_logic_vector(to_unsigned(rs1, 5)) & funct3 &
std_logic_vector(to_unsigned(rd, 5)) & opcode;
        load_inst(inst, signals);
    end procedure op_srl;
```

```
    procedure op_sra(
        constant rd     : in reg_addr;
        constant rs1    : in reg_addr;
        constant rs2    : in reg_addr;

        signal signals  : out std_logic_vector(65 downto 0)
    ) is
        constant funct7     : std_logic_vector(31 downto 25)    := "0100000";
        constant funct3     : std_logic_vector(14 downto 12)    := "101";
        constant opcode     : std_logic_vector(6 downto 0)      := "0110011";
        variable inst       : std_logic_vector(31 downto 0);
    begin
        inst := funct7 & std_logic_vector(to_unsigned(rs2, 5)) &
std_logic_vector(to_unsigned(rs1, 5)) & funct3 &
std_logic_vector(to_unsigned(rd, 5)) & opcode;
        load_inst(inst, signals);
    end procedure op_sra;

    procedure op_or(
        constant rd     : in reg_addr;
        constant rs1    : in reg_addr;
        constant rs2    : in reg_addr;

        signal signals  : out std_logic_vector(65 downto 0)
    ) is
        constant funct7     : std_logic_vector(31 downto 25)    := "0000000";
        constant funct3     : std_logic_vector(14 downto 12)    := "110";
        constant opcode     : std_logic_vector(6 downto 0)      := "0110011";
        variable inst       : std_logic_vector(31 downto 0);
    begin
        inst := funct7 & std_logic_vector(to_unsigned(rs2, 5)) &
std_logic_vector(to_unsigned(rs1, 5)) & funct3 &
std_logic_vector(to_unsigned(rd, 5)) & opcode;
        load_inst(inst, signals);
    end procedure op_or;

    procedure op_and(
        constant rd     : in reg_addr;
        constant rs1    : in reg_addr;
        constant rs2    : in reg_addr;

        signal signals  : out std_logic_vector(65 downto 0)
    ) is
        constant funct7     : std_logic_vector(31 downto 25)    := "0000000";
        constant funct3     : std_logic_vector(14 downto 12)    := "111";
        constant opcode     : std_logic_vector(6 downto 0)      := "0110011";
        variable inst       : std_logic_vector(31 downto 0);
    begin
```

```
        inst := funct7 & std_logic_vector(to_unsigned(rs2, 5)) &
std_logic_vector(to_unsigned(rs1, 5)) & funct3 &
std_logic_vector(to_unsigned(rd, 5)) & opcode;
        load_inst(inst, signals);
    end procedure op_and;
end package body getto_compiler;
```

11.10 GP_Registers_testbench.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity gp_registers_testbench is
end gp_registers_testbench;

-- define the internal organisation and operation of the general-purpose
register testbench
architecture behaviour of gp_registers_testbench is
    -- architecture declarations
    constant clock_delay    : time := 50 ns;

    signal clk, reg_write               : std_logic := '0';
    signal rs1_addr, rs2_addr, rd_addr  : std_logic_vector(4 downto 0) :=
(others => '0');
    signal rd_data                      : std_logic_vector(31 downto 0) :=
(others => '0');
    signal rs1_data, rs2_data           : std_logic_vector(31 downto 0) :=
(others => '0');
-- concurrent statements
begin
    -- instantiate alu_controller
    gp_registers : entity work.u32_gp_registers
    port map (
        clk => clk,
        reg_write => reg_write,
        rs1_addr => rs1_addr,
        rs2_addr => rs2_addr,
        rd_addr => rd_addr,
        rd_data => rd_data,
        rs1_data => rs1_data,
        rs2_data => rs2_data
    );

    process
        procedure test_read(
            constant addr1, addr2   : in std_logic_vector(4 downto 0);
            constant expected1, expected2   : in std_logic_vector(31 downto 0)
        ) is

        begin
            rs1_addr <= addr1;
            rs2_addr <= addr2;
            reg_write <= '0';

            wait for clock_delay;
```

```
        clk <= '1';
        wait for clock_delay;
        clk <= '0';

        assert rs1_data = expected1
        report "Unexpected data: " &
        "rs1_addr = 0x" & to_hex_string(rs1_data) & "; " &
        "rs1_data = " & to_string(rs1_addr) & "; " &
        "expected = 0x" & to_hex_string(expected1) & "; "
        severity error;

        assert rs2_data = expected2
        report "Unexpected data: " &
        "rs1_addr = 0x" & to_hex_string(rs2_data) & "; " &
        "rs1_data = " & to_string(rs2_addr) & "; " &
        "expected = 0x" & to_hex_string(expected2) & "; "
        severity error;
    end procedure test_read;

    procedure test_write(
        constant addr   : in std_logic_vector(4 downto 0);
        constant data   : in std_logic_vector(31 downto 0)
    ) is

    begin
        rd_addr <= addr;
        rd_data <= data;
        reg_write <= '1';

        wait for clock_delay;
        clk <= '1';
        wait for clock_delay;
        clk <= '0';
    end procedure test_write;

    procedure test_rw(
        constant addr   : in std_logic_vector(4 downto 0);
        constant data   : in std_logic_vector(31 downto 0)
    ) is

    begin
        rd_addr <= addr;
        rd_data <= data;
        rs1_addr <= addr;
        reg_write <= '1';

        wait for clock_delay;
        clk <= '1';
```

```
            wait for clock_delay;
            clk <= '0';

            report "Read & Wrote data: " &
            "rd = " & to_string(rd_addr) & "; " &
            "data = 0x" & to_hex_string(rd_data) & "; " &
            "rs1 = 0x" & to_hex_string(rs1_data) & "; "
            severity note;
        end procedure test_rw;

        procedure test_read_all(
            constant expected   : in std_logic_vector(31 downto 0)
        ) is

        begin
            for i in 0 to 15 loop
                if (i = 0) then
                    test_read(std_logic_vector(to_unsigned(i, 5)),
std_logic_vector(to_unsigned((31 - i), 5)), x"00000000", expected);
                else
                    test_read(std_logic_vector(to_unsigned(i, 5)),
std_logic_vector(to_unsigned((31 - i), 5)), expected, expected);
                end if;
            end loop;
        end procedure test_read_all;

        procedure test_write_all(
            constant data   : in std_logic_vector(31 downto 0)
        ) is

        begin
            for i in 0 to 31 loop
                test_write(std_logic_vector(to_unsigned(i, 5)), data);
            end loop;
        end procedure test_write_all;
    begin
        test_read_all(x"00000000");
        test_write_all(x"FFFFFFFF");
        test_read_all(x"FFFFFFFF");
        test_write_all(x"AAAAAAAA");
        test_read_all(x"AAAAAAAA");
        test_write_all(x"55555555");
        test_read_all(x"55555555");
        test_write_all(x"00000000");
        test_read_all(x"00000000");
        wait;
    end process;
end behaviour;
```

11.11 GP_Registers_testbench.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;

entity immediate_decoder_testbench is
end immediate_decoder_testbench;

-- define the internal organisation and operation of the immediate decoder
testbench
architecture behaviour of immediate_decoder_testbench is
    -- architecture declarations
    constant time_delta : time := 100 ns;

    signal inst     : std_logic_vector(31 downto 2) := (others => '0');
    signal imm      : std_logic_vector(31 downto 0) := (others => '0');

-- concurrent statements
begin
    -- instantiate alu
    u32_immediate_decoder : entity work.u32_immediate_decoder
    port map (
        inst => inst,
        imm => imm
    );

    process
        procedure test(
            constant inst_type      : in string;
            constant opcode         : in std_logic_vector(6 downto 2);
            constant instruction    : in std_logic_vector(31 downto 2);
            constant expected       : in integer
        ) is
            variable res : integer;
        begin
            -- assign values to circuit inputs
            inst <= instruction;

            wait for time_delta;

            res := conv_integer(imm);

            assert res = expected
            report "unexpected result: " &
            "instruction_type = " & inst_type & "; " &
            "opcode = " & to_string(opcode) & "; " &
            "result = " & integer'image(res) & "; " &
            "expected = " & integer'image(expected)
```

```
        severity error;
    end procedure test;

    procedure test_itype(
        constant immediate  : in std_logic_vector(31 downto 20);
        constant expected   : in integer
    ) is
        constant inst_type  : string := "i";
        variable opcode     : std_logic_vector(6 downto 2);
        variable inst       : std_logic_vector(31 downto 2);
    begin
        opcode := "00100";
        inst := immediate & (19 downto 7 => '1') & opcode;
        test(inst_type, opcode, inst, expected);
        opcode := "11001";
        inst := immediate & (19 downto 7 => '1') & opcode;
        test(inst_type, opcode, inst, expected);
        opcode := "00000";
        inst := immediate & (19 downto 7 => '1') & opcode;
        test(inst_type, opcode, inst, expected);
    end procedure test_itype;

    procedure test_utype(
        constant immediate  : in std_logic_vector(31 downto 12);
        constant expected   : in std_logic_vector(31 downto 0)
    ) is
        constant inst_type  : string := "u";
        variable opcode     : std_logic_vector(6 downto 2);
        variable inst       : std_logic_vector(31 downto 2);
    begin
        opcode := "01101";
        inst := immediate & (11 downto 7 => '1') & opcode;
        test(inst_type, opcode, inst, conv_integer(expected));
        opcode := "00101";
        inst := immediate & (11 downto 7 => '1') & opcode;
        test(inst_type, opcode, inst, conv_integer(expected));
    end procedure test_utype;

    procedure test_jtype(
        constant immediate  : in std_logic_vector(31 downto 12);
        constant expected   : in std_logic_vector(31 downto 0)
    ) is
        constant inst_type  : string := "j";
        variable opcode     : std_logic_vector(6 downto 2);
        variable inst       : std_logic_vector(31 downto 2);
    begin
        opcode := "11011";
        inst := immediate & (11 downto 7 => '1') & opcode;
```

```
            test(inst_type, opcode, inst, conv_integer(expected));
        end procedure test_jtype;

        procedure test_btype(
            constant immediate  : in std_logic_vector(11 downto 0);
            constant expected   : in std_logic_vector(31 downto 0)
        ) is
            constant inst_type  : string := "b";
            variable opcode     : std_logic_vector(6 downto 2);
            variable inst       : std_logic_vector(31 downto 2);
        begin
            opcode := "11000";
            inst := immediate(11 downto 5) & (24 downto 12 => '1') &
immediate(4 downto 0) & opcode;
            test(inst_type, opcode, inst, conv_integer(expected));
        end procedure test_btype;

        procedure test_stype(
            constant immediate  : in std_logic_vector(11 downto 0);
            constant expected   : in integer
        ) is
            constant inst_type  : string := "s";
            variable opcode     : std_logic_vector(6 downto 2);
            variable inst       : std_logic_vector(31 downto 2);
        begin
            opcode := "01000";
            inst := immediate(11 downto 5) & (24 downto 12 => '1') &
immediate(4 downto 0) & opcode;
            test(inst_type, opcode, inst, expected);
        end procedure test_stype;
    begin
        test_itype(x"000", 0);
        test_itype(x"aaa", -1366);
        test_itype(x"555", 1365);
        test_itype(x"fff", -1);

        test_utype(x"00000", x"00000000");
        test_utype(x"aaaaa", x"aaaaa000");
        test_utype(x"55555", x"55555000");
        test_utype(x"fffff", x"fffff000");

        test_jtype(x"00000", x"00000000");
        test_jtype(x"aaaaa", x"fffaa2aa");
        test_jtype(x"55555", x"00055d54");
        test_jtype(x"fffff", x"fffffffe");

        test_btype(x"000", x"00000000");
        test_btype(x"aaa", x"fffff2aa");
```

```
        test_btype(x"555", x"00000d54");
        test_btype(x"fff", x"fffffffe");

        test_stype(x"000", 0);
        test_stype(x"aaa", -1366);
        test_stype(x"555", 1365);
        test_stype(x"fff", -1);
        wait;
    end process;
end behaviour;
```

11.12 GP_Registers_testbench.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;

entity inst_fetch_testbench is
end inst_fetch_testbench;

-- define the internal organisation and operation of the instruction fetch
pipeline stage testbench
architecture behaviour of inst_fetch_testbench is
    -- architecture declarations
    constant clock_delay    : time := 50 ns;

    signal clk, write_en, pc_src            : std_logic      := '0';
    signal write_inst, write_addr, new_pc,
            inst, pc_out, next_pc_out       : std_logic_vector(31 downto 0)
:= (others => '0');
    signal current_pc                       : std_logic_vector(31 downto 0)
:= (others => '0');
-- concurrent statements
begin
    -- instantiate instruction fetch pipeline stage
    inst_fetch : entity work.u32_inst_fetch
    port map (
        clk => clk,
        write_en => write_en,
        pc_src => pc_src,
        write_inst => write_inst,
        write_addr => write_addr,
        new_pc => new_pc,
        inst => inst,
        pc_out => pc_out,
        next_pc_out => next_pc_out
    );

    process
        procedure test(
            constant jump   : in std_logic_vector(31 downto 0);
            constant source : in std_logic
        ) is
            variable expected   : std_logic_vector(31 downto 0) := (others =>
'0');
        begin
            pc_src <= source;
            new_pc <= jump;
            write_en <= '0';
```

```
        wait for clock_delay;
        clk <= '1';
        wait for clock_delay;
        clk <= '0';

        wait for 1 ps;

        expected := x"000000FC" - current_pc;

        assert (pc_out = current_pc)
        report "Unexcpected PC: " &
        "pc = 0x" & to_hex_string(pc_out) & "; " &
        "pc_src = " & to_string(pc_src) & "; " &
        "expected = 0x" & to_hex_string(current_pc) & "; "
        severity error;

        assert (inst = expected)
        report "Unexcpected result: " &
        "inst = 0x" & to_hex_string(inst) & "; " &
        "expected = 0x" & to_hex_string(expected) & "; "
        severity error;

        if source = '1' then
            current_pc <= jump;
        else
            current_pc  <= current_pc + x"00000004";
        end if;
    end procedure test;

    procedure fill_inst_mem is begin
        for i in 0 to 63 loop
            write_en <= '1';
            write_inst <= std_logic_vector(to_unsigned(i * 4, 32));
            write_addr <= std_logic_vector(to_unsigned((63 - i) * 4, 32));

            wait for clock_delay;
            clk <= '1';
            wait for clock_delay;
            clk <= '0';
        end loop;

        wait for 1 ps;
        assert (pc_out = x"00000000")
        report "Unexcpected PC: " &
        "pc = 0x" & to_hex_string(pc_out) & "; " &
        "pc_src = " & to_string(pc_src) & "; " &
        "expected = 0x00000000; "
```

```
            severity error;

            assert (inst = x"000000FC")
            report "Unexcpected result: " &
            "inst = 0x" & to_hex_string(inst) & "; " &
            "expected = 0x000000FC; "
            severity error;

            current_pc  <= current_pc + x"00000004";
        end procedure fill_inst_mem;
    begin
        -- test vectors set based on data mem size of 256B
        fill_inst_mem;

        test(x"00000000", '0');
        test(x"00000024", '1');
        test(x"00000000", '0');
        wait for 10 ns;
        wait;
    end process;
end behaviour;
```

11.13 Inst_Fetch_testbench.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;

entity inst_fetch_testbench is
end inst_fetch_testbench;

-- define the internal organisation and operation of the instruction fetch
pipeline stage testbench
architecture behaviour of inst_fetch_testbench is
    -- architecture declarations
    constant clock_delay    : time := 50 ns;

    signal clk, write_en, pc_src            : std_logic      := '0';
    signal write_inst, write_addr, new_pc,
            inst, pc_out, next_pc_out       : std_logic_vector(31 downto 0)
:= (others => '0');
    signal current_pc                       : std_logic_vector(31 downto 0)
:= (others => '0');
-- concurrent statements
begin
    -- instantiate instruction fetch pipeline stage
    inst_fetch : entity work.u32_inst_fetch
    port map (
        clk => clk,
        write_en => write_en,
        pc_src => pc_src,
        write_inst => write_inst,
        write_addr => write_addr,
        new_pc => new_pc,
        inst => inst,
        pc_out => pc_out,
        next_pc_out => next_pc_out
    );

    process
        procedure test(
            constant jump   : in std_logic_vector(31 downto 0);
            constant source : in std_logic
        ) is
            variable expected   : std_logic_vector(31 downto 0) := (others =>
'0');
        begin
            pc_src <= source;
            new_pc <= jump;
            write_en <= '0';
```

```
        wait for clock_delay;
        clk <= '1';
        wait for clock_delay;
        clk <= '0';

        wait for 1 ps;

        expected := x"000000FC" - current_pc;

        assert (pc_out = current_pc)
        report "Unexcpected PC: " &
        "pc = 0x" & to_hex_string(pc_out) & "; " &
        "pc_src = " & to_string(pc_src) & "; " &
        "expected = 0x" & to_hex_string(current_pc) & "; "
        severity error;

        assert (inst = expected)
        report "Unexcpected result: " &
        "inst = 0x" & to_hex_string(inst) & "; " &
        "expected = 0x" & to_hex_string(expected) & "; "
        severity error;

        if source = '1' then
            current_pc <= jump;
        else
            current_pc  <= current_pc + x"00000004";
        end if;
    end procedure test;

    procedure fill_inst_mem is begin
        for i in 0 to 63 loop
            write_en <= '1';
            write_inst <= std_logic_vector(to_unsigned(i * 4, 32));
            write_addr <= std_logic_vector(to_unsigned((63 - i) * 4, 32));

            wait for clock_delay;
            clk <= '1';
            wait for clock_delay;
            clk <= '0';
        end loop;

        wait for 1 ps;
        assert (pc_out = x"00000000")
        report "Unexcpected PC: " &
        "pc = 0x" & to_hex_string(pc_out) & "; " &
        "pc_src = " & to_string(pc_src) & "; " &
        "expected = 0x00000000; "
```

```
            severity error;

            assert (inst = x"000000FC")
            report "Unexcpected result: " &
            "inst = 0x" & to_hex_string(inst) & "; " &
            "expected = 0x000000FC; "
            severity error;

            current_pc  <= current_pc + x"00000004";
        end procedure fill_inst_mem;
    begin
        -- test vectors set based on data mem size of 256B
        fill_inst_mem;

        test(x"00000000", '0');
        test(x"00000024", '1');
        test(x"00000000", '0');
        wait for 10 ns;
        wait;
    end process;
end behaviour;
```

11.14 Mem_Access_testbench.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity mem_access_testbench is
end mem_access_testbench;

-- define the internal organisation and operation of the memory access
pipeline stage testbench
architecture behaviour of mem_access_testbench is
    -- architecture declarations
    constant clock_delay    : time := 50 ns;

    signal clk, clk_en                          : std_logic
:= '0';
    signal control, rd, rd_out                  : std_logic_vector(4 downto 0)
:= (others => '0');
    signal funct                                : std_logic_vector(3 downto 0)
:= (others => '0');
    signal alu_result, add_result, addr_const,
                oper, funct_rdd                 : std_logic_vector(31 downto
0) := (others => '0');
    signal control_out                          : std_logic_vector(1 downto 0)
:= (others => '0');
-- concurrent statements
begin
    -- instantiate u32 memory access
    mem_access : entity work.u32_mem_access
    port map (
        clk => clk,
        clk_en => clk_en,
        control => control,
        rd => rd,
        funct => funct,
        alu_result => alu_result,
        add_result => add_result,
        addr_const => addr_const,
        oper => oper,
        funct_rdd => funct_rdd,
        rd_out => rd_out,
        control_out => control_out
    );

    process
        procedure test(
            constant passed_alu_result, passed_add_result,
```

```
                        passed_addr_const                   : in
std_logic_vector(31 downto 0);
            constant passed_funct                           : in
std_logic_vector(3 downto 0);
            constant inst_type                              : in string
        ) is
        begin
            alu_result <= passed_alu_result;
            add_result <= passed_add_result;
            addr_const <= passed_addr_const;
            funct <= passed_funct;
            rd <= "11111";
            clk_en <= '1';

            wait for clock_delay;
            clk <= '1';
            wait for clock_delay;
            clk <= '0';

            wait for 1 ps;
            assert rd_out = "11111"
            report "Unexcpected result: " &
            "instruction type = " & inst_type & "; " &
            "rd_out = " & to_string(rd_out) & "; " &
            "expected = 11111; "
            severity error;
        end procedure test;

        procedure test_lui_jal_jalr is
            constant inst_type : string := "LUI";
        begin
            control <= "10000";
            test(x"FFFFFFFF", x"000000AA", x"55555555", "1111", inst_type);
            assert funct_rdd = x"55555555"
            report "Unexcpected result: " &
            "instruction type = " & inst_type & "; " &
            "funct_rdd = " & to_hex_string(funct_rdd) & "; " &
            "expected = 0x55555555; "
            severity error;
        end procedure test_lui_jal_jalr;

        procedure test_auipc is
            constant inst_type : string := "AUIPC";
        begin
            control <= "10001";
            test(x"FFFFFFFF", x"000000AA", x"55555555", "1111", inst_type);
            assert funct_rdd = x"000000AA"
            report "Unexcpected result: " &
```

```
            "instruction type = " & inst_type & "; " &
            "funct_rdd = " & to_hex_string(funct_rdd) & "; " &
            "expected = 0x000000AA; "
            severity error;
        end procedure test_auipc;

        procedure test_load(
            constant passed_add_result, expected  : in std_logic_vector(31
downto 0)
        ) is
            constant inst_type : string := "LOAD";
        begin
            control <= "11011";
            test(x"FFFFFFFF", passed_add_result, x"55555555", "1110",
inst_type);
            assert oper = expected
            report "Unexcpected result: " &
            "instruction type = " & inst_type & "; " &
            "addr = " & to_hex_string(passed_add_result) & "; " &
            "oper = 0x" & to_hex_string(oper) & "; " &
            "expected = 0x" & to_hex_string(expected) & "; "
            severity error;
            assert funct_rdd = x"0000000E"
            report "Unexcpected result: " &
            "instruction type = " & inst_type & "; " &
            "funct_rdd = " & to_hex_string(funct_rdd) & "; " &
            "expected = 0x0000000E; "
            severity error;
        end procedure test_load;

        procedure test_store(
            constant passed_add_result, passed_alu_result : in
std_logic_vector(31 downto 0)
        ) is
            constant inst_type : string := "STORE";
        begin
            control <= "00101";
            test(passed_alu_result, passed_add_result, x"55555555", "1110",
inst_type);
        end procedure test_store;

        procedure test_op_opimm is
            constant inst_type : string := "OP";
        begin
            control <= "10010";
            test(x"FFFFFFFF", x"000000AA", x"55555555", "1111", inst_type);
            assert funct_rdd = x"FFFFFFFF"
            report "Unexcpected result: " &
```

```
            "instruction type = " & inst_type & "; " &
            "funct_rdd = " & to_hex_string(funct_rdd) & "; " &
            "expected = 0xFFFFFFFF; "
            severity error;
        end procedure test_op_opimm;
    begin
        test_lui_jal_jalr; -- check the addr_const is propgated output of
funct_rdd

        test_auipc; -- check the add_result is propgated output of funct_rdd

        test_op_opimm; -- check the alu_result is propgated output of
funct_rdd

        test_load(x"00000004", x"00000000"); -- check no data stored in byte
address 4

        test_store(x"00000004", x"AAAAAAAA"); -- store data in byte address 4

        test_load(x"00000004", x"AAAAAAAA"); -- check data stored in byte
address 4
        wait for 10 ns;
        wait;
    end process;
end behaviour;
```